SERIAL PORT COMPLETE

SECOND EDITION

Design serial links and networks

Access devices with .NET

Program microcontrollers for serial communications

Use wireless technologies

COM PORTS, USB VIRTUAL COM PORTS AND PORTS FOR EMBEDDED SYSTEMS

JAN AXELSON author of USB COMPLETE

Serial Port Complete

COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems

Second Edition

Jan Axelson

Lakeview Research LLC
Madison, WI 53704

Serial Port Complete:
COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems
Second Edition

Jan Axelson

Published by Lakeview Research LLC, 5310 Chinook Ln., Madison WI 53704

On the web at **www.Lvr.com**

Distributed by Independent Publishers Group (www.ipgbook.com).

14 13 12 11 10 9 8 7 6 5 4 3 2 1

ISBN 978-1931448-07-9

Print ISBN: 9781931448062

Contents

This page intentionally left blank

Introduction

When I wrote the first edition of this book, the RS-232 serial port was the workhorse of PC interfaces. Modems and scores of other peripherals connected to PCs via the serial ports that were present on every machine.

When the Universal Serial Bus (USB) took hold in the late 1990s, many predicted that serial ports would soon be obsolete. Plenty of peripherals that formerly used the serial port have switched to USB. But some devices can't use USB or have requirements that USB alone can't provide. Many embedded systems use serial ports because they're inexpensive and less complex to program compared to USB. Serial ports can use longer cables than USB allows. And the RS-485 serial interface supports networks suitable for many monitoring and control applications.

While most PCs no longer have built-in serial (COM) ports, the ports are easy to add via USB converters. With converters, the number of expansion slots no longer limits the number of serial ports a system can have. The SerialPort class included in Microsoft's .NET Framework shows that PC applications continue to find COM-port communications useful.

What's Inside

This book explores wide and varied territory, including hardware and software; ports in PCs and in embedded systems; and RS-232, RS-485, and wireless interfaces. You don't need to read the book straight through. If you're interested in a particular topic, you can skip right to it.

The first chapters focus on hardware and interfacing. Chapters 1–2 are an introduction to asynchronous serial communications. Chapter 3 discusses serial ports in PCs, and chapters 4–8 are a guide to interfacing using RS-232, RS-485, and wireless technologies.

The next chapters are a guide to programming. Chapters 9–10 show how to program serial ports on PCs using Visual Basic .NET and Visual C# .NET. Chapter 11 shows how to program serial ports for embedded systems with examples for microEngineering Labs's PICBASIC PRO compiler and Microchip Technology's MPLAB® C18 C compiler.

Chapters 12–13 focus on hardware and programming for RS-485 serial networks.

Chapters 14–16 explain how to implement USB virtual COM ports using special-purpose and generic USB controllers.

If you're looking for example code, see the entries under *code example (embedded)* and *code example (PC)* in the index.

What's New in the Second Edition

Much has happened in the world of computing since the first edition of this book was released. For this second edition, I've revised and updated the contents from start to finish.

One addition is example code in C/C# as well as Basic. This book includes code examples for PCs and for embedded systems (microcontrollers).

Also new in the Second Edition are these topics:

- Designing and programming USB virtual COM ports.
- Using wireless technologies to transmit serial data.
- Accessing serial ports over Ethernet or Wi-Fi networks.
- Transferring any kind of text data using Unicode encoding.

Who Should Read this Book?

Whether your interest is hardware or software and whether you work with PCs, embedded systems, or both, you'll find useful guidance in this book.

Programmers will learn how to communicate via serial ports, including USB virtual COM ports, in PCs and embedded systems. The example code for PCs and microcontrollers in Basic and C/C# provides a quick start for a variety of applications.

Circuit designers will find designs for a variety of applications including converters that translate between RS-232, RS-485, and 3V/5V logic. Designs with fail-safe features, high noise immunity, and low power consumption are included.

Hobbyists and experimenters will find inspiration for projects.

Teachers and students can learn about serial ports and use the examples in this book to demonstrate concepts.

This book assumes you have a basic knowledge of electronics and either Basic/Visual Basic or C/C# programming. I assume no previous knowledge or experience with serial-port hardware or programming.

Example Code and Updates

At the start of each code example, a sidehead indicates the programming language used:

Sidehead	Programming Language	Provider
VB	Visual Basic .NET	Microsoft
VC#	Visual C# .NET	Microsoft
PBP	PICBASIC PRO	microEngineering Labs, Inc.
C18	MPLAB C18 compiler	Microchip Technology Inc.

Example applications are available for free download from *www.Lvr.com*. This is also the place to find updates, corrections, and other links to information and tools for serial-port applications.

Abbreviations

This book uses the following abbreviations to express quantities and units:

Multipliers

Abbreviation	Description	Value
p	pico	10^{-12}
n	nano	10^{-9}
µ	micro	10^{-6}
m	milli	10^{-3}
k	kilo	1^{3}
M	mega	10^{6}

Electrical

Abbreviation	Meaning
A	amperes
F	farads
Ω	ohms
V	volts

Time

Abbreviation	Meaning
s	seconds
hr	hours
Hz	Hertz (cycles per second)

Distance

Abbreviation	Meaning
in.	inches
ft	feet

bps = bits per second

Some expressions contain multiple units, such as ps (picoseconds) and mA (milliamperes).

Number Systems

The following conventions apply to numeric values in the text:

Binary values have a trailing subscript “b”. Example: $10100011_{b.}$

Hexadecimal values have a trailing “h”. Example: A3h

All other values are decimal. Example: 163

This page intentionally left blank

Acknowledgments

First, I want to thank the readers of *Serial Port Complete*'s first edition and the readers of my other books and articles. This book is much improved due to the the many suggestions and comments I've received from readers over the years.

For help in preparing the second edition, I thank my technical reviewers: John Hyde for his generosity, encouragement, and good suggestions; Rawin Rojvanit for once again providing a thoughtful and expert critique; and Tsuneo Chinzei for sharing his knowledge on USB virtual COM ports. Thanks to Ron Smith for wonderful circuits and great conversations about RS-485. Thanks to Steve Drake for helping me see my writing through the eyes of a typical reader. And thanks to Jim Hughes for taking good photos.

This book is dedicated to Michele, Pat, and Isie.

1

Options and Choices

A serial port is a computer interface that transmits data one bit at a time. In common use, the term "serial port" refers to ports that use a particular asynchronous protocol. These ports include the RS-232 ports on PCs and many serial ports in embedded systems. Most serial ports are bidirectional: they can both send and receive data. Transmitting one bit at a time might seem inefficient but has advantages, including the ability to use inexpensive cables and small connectors.

In PCs, applications access most serial ports as COM ports. Applications that use Microsoft's .NET Framework class library can use the SerialPort class to access COM ports. Some USB devices function as virtual COM ports, which applications can access in the same way as physical serial ports. Some Ethernet and Wi-Fi devices function as serial servers that enable applications to access serial ports over a network.

Microcontrollers in embedded systems can use serial ports to communicate with other embedded systems and PCs. Language compilers for microcontrollers often provide libraries with functions that simplify serial-port programming.

When to use a Serial Port

Device developers have many options for computer interfaces. Table 1-1 compares popular wired interfaces.

Serial ports are ideal for many communications between embedded systems or between embedded systems and PCs. Serial ports can also be a good choice when you need very long cables or a basic network among PCs, embedded systems, or a combination. Some systems include a serial port that is hidden from users but available to technicians for debugging and diagnostics.

Advantages

These are some advantages of asynchronous serial ports and COM-port programming:

- Serial ports can exchange just about any type of information. Applications suited for serial ports often involve reading sensors, switches, or other inputs or controlling motors, relays, displays, or other outputs.
- The hardware is inexpensive and readily available. PCs that don't have built-in serial ports can use USB/serial converters. Just about every microcontroller family includes variants with built-in serial ports.
- Other than the Start, Stop, and optional parity bits added to each transmitted byte, serial interfaces assume nothing about the content of the data being transmitted. In contrast, USB and Ethernet use sophisticated protocols that define the format of transmitted data. Hardware or firmware must implement these protocols, adding complexity that some applications don't need.
- Cables can be very long. An RS-232 interface can use cables of 130 ft or more. An RS-485 cable can be over 4000 ft. In contrast, the maximum distance between a USB device and its host is 16 ft, or 98 ft with five hubs. Ethernet cables have a maximum length of 328 ft.
- The cables are inexpensive. Many links can use unshielded cables with 3–9 wires.
- For devices that connect to PCs, Windows and other operating systems provide drivers for accessing COM ports. Programming languages provide classes, libraries, or other tools for COM-port communications.

Table 1-1: Comparison of popular computer interfaces. Where a standard doesn't specify a maximum, the table shows a typical maximum.

Interface	Format	Number of Devices (maximum)	Distance (maximum, ft)	Speed (maximum, bps)	Typical Use
RS-232 (TIA-232)	asynchronous serial	2	50-100	20k (faster with some hardware)	Modem, basic communications
RS-485 (TIA-485)	asynchronous serial	32 unit loads (up to 256 devices with some hardware)	4000	10M	Data acquisition and control systems
Ethernet	serial	1024	1600	10G	PC network communications
IEEE-1394b (FireWire 800)	serial	64	300	3.2G	Video, mass storage
IEEE-488 (GPIB)	parallel	15	60	8M	Instrumentation
I²C	synchronous serial	40	18	3.4M	Microcontroller communications
Microwire	synchronous serial	8	10	2M	Microcontroller communications
MIDI	serial current loop	2 (more with flow-through mode)	50	31.5k	Music, show control
Parallel Printer Port	parallel	2 (8 with daisy-chain support)	10–30	8M	Printer
SPI	synchronous serial	8	10	2.1M	Microcontroller communications
USB	asynchronous serial	127	16 (up to 98 ft with 5 hubs)	1.5M, 12M, 480M	PC peripherals

- A USB device accessed as a COM port doesn't have to have an asynchronous serial interface. The device can have a parallel or other interface as needed to suit the application.
- Wireless technologies enable transmitting serial data without cables.

Limits

No single interface is ideal for every purpose. Limits to asynchronous serial interfaces include these:

- The computers at each end must convert between the transmitted serial data and the CPU's parallel data bus. The conversion is usually handled automatically by hardware, however.
- The specified maximum bit rate for RS-232 is 20 kbps. But many interface chips can exceed this rate, and RS-485 supports speeds of up to 10 Mbps. Communications between a PC and a USB Virtual COM ports aren't limited by RS-232's maximum bit rate.
- Windows doesn't promise real-time performance for serial communications. Sending or receiving data may need to wait as the operating system attends to other tasks. But the delays are normally short and are common to other interfaces on Windows systems. Embedded systems typically can control the scheduling of serial communications more precisely.

System Components

Communicating via serial ports requires three things: computers with serial ports, a cable or wireless interface that provides a physical link between the ports, and programming to manage the communications.

The Computers

Just about any computer can use serial-port communications, including inexpensive microcontrollers and PCs that don't have built-in serial ports.

Examples of Serial Ports

Devices with asynchronous serial ports typically contain a hardware component called a Universal Asynchronous Transmitter/Receiver (UART). The UART converts between parallel and serial data and handles other low-level details of serial communications.

These are some examples of ports controlled by UARTs:

- Microcontroller serial ports. Many microcontrollers contain one or more UARTs for serial-port communications. When a hardware UART isn't available, microcontroller firmware can emulate a UART's functions, typically with assistance from an on-chip timer.
- External UART chips that interface to microcontrollers or other CPUs.
- The RS-232 serial ports that were standard on PCs and other devices before USB became common. Each of these ports contains a UART that interfaces to the system's CPU. Any PC with a free expansion slot can add this type of port on an expansion card.
- RS-232 ports on PC Cards (also called PCMCIA cards). Any PC with a free PC-Card slot can use these.
- Serial ports that connect to PCs via USB converter modules.
- Other serial ports used in long-distance and networking applications, often in industrial-control applications. These interfaces include RS-485, RS-422, and RS-423. Expansion cards, PC Cards, and USB converters with these interfaces are available.
- Ports on serial-server modules that connect to Ethernet or Wi-Fi networks.

On PCs, ports that applications can access as COM ports include these:

- RS-232 ports on older motherboards or on expansion cards.
- Ports that connect to a PC via a USB converter that uses a driver that assigns a COM port to the device. Converters are available as modules and as chips for incorporating into circuits. A converter can convert between USB and RS-232, RS-485, TTL serial, or even a parallel interface.
- Internal modems that interface to phone lines.
- Serial ports on network serial-server modules.

For USB virtual COM-port devices, Windows includes a driver for USB's communication devices class (CDC). For improved performance, some converters use vendor-specific drivers in place of the provided Windows drivers.

Computer Types

Computers that communicate via serial ports don't have to be all the same type. Tiny microcontrollers can talk to the latest PCs as long as both ends of the link use compatible interfaces and protocols. The PC examples in this book are for the family of computers that has evolved from the IBM PC, including desktop

and notebook PCs. Other computer types also have serial ports that are built in or available via converters or expansion cards.

An embedded system is a computer-controlled device dedicated to performing a single task or a set of related tasks. Embedded systems are typically built into, or embedded in, the devices they control. For example, a modem is an embedded system that handles tasks of data communications over the phone system. Some embedded systems are one-of-a-kind or small-quantity projects. Many involve monitoring or control tasks.

Embedded systems often use microcontrollers, which contain a CPU and I/O hardware such as UARTs. Microcontroller chips can be classified by data-bus width: 8-bit chips have an 8-bit data path and are popular in monitoring and control applications. Chips with 4-, 16-, and 32-, and 64-bit data buses are also available. Different chips have different combinations of features and abilities, including asynchronous and synchronous serial ports, USB controllers, type and amount of memory for storing programs and data, and support for power-saving modes.

The Physical Link

The physical link between computers consists of the wires or other medium that carries information from one computer to another and the connectors and other components that interface the medium to the computers.

RS-232 links can use just about any cable type and require one line per signal plus a common ground line. RS-485 networks typically use twisted-pair cables with a pair for each differential signal. Other options for serial communications include fiber-optic cable, which encodes data as the presence or absence of light, and wireless technologies, which enable sending data as electromagnetic (radio) or infrared signals through the air.

Computers connected by wires must have a common ground reference, typically provided by a ground wire in the cable.

Programming

A computer must perform the following tasks in serial communications:

- Detect and process received data.
- Provide and send data as needed.
- Carry out any other tasks the computer is responsible for.

If the connection is to a serial network, each computer must ignore communications intended for other computers in the network and comply with network protocols for addressing transmitted data to the appropriate computer(s).

Program code carries out these tasks, often with help from hardware.

Languages

The programming for a serial interface can use any language, and the language doesn't have to be the same on every computer. The only requirement is that all of the computers must agree on a format. Microcontroller programs might access UART registers directly or use library functions or other higher-level methods to set communications parameters and exchange data. PC applications typically use higher-level functions to access ports.

Protocols

A protocol is a set of rules that defines how computers manage communications. Serial communications must implement a low-level communication protocol and may also implement a higher-level message protocol.

Communication Protocols

A communication protocol defines how the bits travel, including when a computer can transmit, the bit rate, and in what order the bits transmit. The UART typically handles the details of sending individual bits and storing received bits on the serial port.

Two computers that want to exchange data must agree on whether both ends can transmit at once or whether the computers need to take turns. Most wired links between two computers are full duplex: both computers can transmit at the same time. Many wireless links are half duplex: the computers must take turns. A simplex link is one way only. A network with three or more computers sharing a data path must use a protocol that defines when a computer can transmit.

A communication protocol can include the use of status and control lines. These lines can indicate when a transmitter has data to send or when a receiver is able to accept new data. The process of exchanging this information is called flow control. Hardware flow control uses dedicated lines for the signals. Devices can also use software flow control to provide the same information by sending defined codes, typically in the same path used for data.

Additional status and control lines can provide other information such as the presence of a carrier frequency or a ring signal on a phone line. In serial networks where only one transmitter can be enabled at a time, a transmit-enable line at each computer can enable and disable the transmitters as needed.

Message Protocols

Serial communications often exchange messages that consist of blocks of data with defined formats. A message protocol can specify what type of data a message contains and how information is structured within the message.

The computers in a network need a way to detect which computer is the intended receiver of transmitted data. Networks typically assign an address to each computer and include the receiver's address in each message. For example, a very basic message might consist of two bytes: one byte to identify the receiver and one byte containing data.

To enable a receiving computer to detect the start and end of a message, a message can include codes to indicate these events or a header that stores the message length. A message can also include one or more bytes that the receiving computer uses in error checking.

Applications

One way to think about serial applications is by the primary direction of data flow. In a link between two computers, one computer might gather data from or send commands to the other computer. Or two computers may each be responsible for various monitoring and control functions, sharing information with each other.

In some systems all computers send and receive more or less equally. In others, most of the data flows to or from a central computer. For example, most of the activity in a network might involve one computer that collects data from computers in remote locations.

Example Systems

An everyday example of a system that collects data is a weather-watching network. A desktop PC might serve as a primary computer that controls the activities of one or more secondary computers, which can be embedded systems or PCs. The primary computer sends commands to the secondary computers to tell them how often to collect data, what data to send, and when to send. The data collected might include temperature, air pressure, rainfall, and so on. At

intervals, each secondary computer sends its collected data to the primary computer, which stores the data and makes it available for viewing and processing. This basic setup is adaptable to many other types of data-gathering systems.

Other systems are mainly concerned with controlling external devices, rather than gathering data from them. A store-window display might include a set of small robots, each with switches and signals that control motors, lights, and other mechanical or electrical devices. Each robot is an embedded system, and a primary computer controls the show by sending commands to the robots. The robots can also return information about their current states, but the main job of this type of system is to control devices, rather than to collect information from them.

An example of a system involved with both monitoring and controlling is a home-control system that monitors temperature, humidity, motion, switch states, and other conditions throughout a house. Other circuits control the home's heating, cooling, lighting, audio and video systems, and alarms. When the data (or a lack of data) indicates a problem, the system generates an alarm.

Managing Communications

In each of the examples above, one computer typically acts as a primary computer that controls a series of secondary computers. A secondary computer transmits only after the primary computer contacts it and gives it permission.

Some networks have no primary computer. Instead, each computer has equal status with the others, and each can request actions from the others. For example, each computer might transmit in a defined sequence. Or on receiving a message, a computer might have permission to select any other computer to transmit to.

Special-purpose Modules

Many common peripheral functions are available as modules with serial interfaces. These modules make it easy to add a function to a design. For example, LCD modules with serial interfaces are available from Scott Edwards Electronics (*www.seetron.com*). The USBwiz from GHI Electronics (*www.ghielectronics.com*) contains a USB host controller and makes it possible to access USB devices via an asynchronous serial port. Motor controllers with serial interfaces are also available from a variety of sources.

These are just a few examples. This book will guide you in choosing components and writing programs for whatever serial-port application you have in mind.

2

Formats and Protocols

This chapter introduces formats and protocols used in asynchronous serial communications including low-level data formats, encoding methods for binary and text data, and protocols to ensure reliable data transfer.

Sending Serial Data

A serial port output that functions as a transmitter, or driver, sends bits one at a time to a serial-port input that functions as a receiver, typically on a different computer. The cable between the computers typically has a dedicated data path for each direction. Some serial interfaces have a single, shared data path for both directions, with the transmitters taking turns.

Asynchronous and Synchronous Communications

The serial communications described in this book use an asynchronous protocol. In an asynchronous protocol, the interface doesn't include a clock line. Instead, each computer provides its own clock to use as a timing reference. The computers must agree on a clock frequency, and the actual frequencies at each

computer must match within a few percent. A transmitted Start bit synchronizes the transmitter's and receiver's clocks.

In contrast, in a synchronous protocol, the interface includes a clock line typically controlled by one of the computers, and all transmitted bits synchronize to that clock. Each transmitted bit is valid at a defined time after a clock's rising or falling edge, depending on the protocol. Examples of synchronous serial interfaces are I^2C, SPI, and Microwire.

Word Formats

A UART transmits data in chunks often called words. Each word contains a Start bit, data bits, an optional parity bit, and one or more Stop bits.

Most UARTs support multiple word formats. A common format is 8-N-1, where the transmitter sends each word as one Start bit, followed by eight data bits and one Stop bit. The data bits transmit beginning with bit 0 (the least significant bit, or LSb). Figure 2-1 illustrates.

The N in 8-N-1 indicates that the words don't contain a parity bit. Formats that use the parity bit can use a parity type of even, odd, mark, or space. An example format using parity is 7-E-1 where the transmitter sends one Start bit, seven data bits, an even parity bit, and one Stop bit.

The parity bit can provide a basic form of error detecting. With even parity, the data bits and parity bit in each word contain an even number of 1s. With odd parity, the data bits and parity bit in each word contain an odd number of 1s. For example, assume the data bits to send are 0000001_b. With even parity, the parity bit is 1 to bring the total number of 1s in the data and parity bits to an even number. With odd parity, the parity bit is 0 to keep the total number of 1s odd. If the data bits are 0000011_b, with even parity, the parity bit is 0, and with odd parity, the parity bit is 1. If the communications are using 7-E-1 format, a receiving computer that receives a byte with an odd number of 1s knows the data didn't transmit correctly.

Mark and space parity are forms of stick parity. With mark parity, the parity bit is always 1, and with space parity, the parity bit is always zero. One use for these parities is in 9-bit networks that use mark and space bits to indicate whether the data bits contain an address or data. Chapter 12 has more about 9-bit networks.

Most UARTs support 7- and 8-bit data. Some UARTs support anywhere from 5 to 8 data bits. The data bits in a transmitted word are sometimes referred to as a character and may in fact represent a text character.

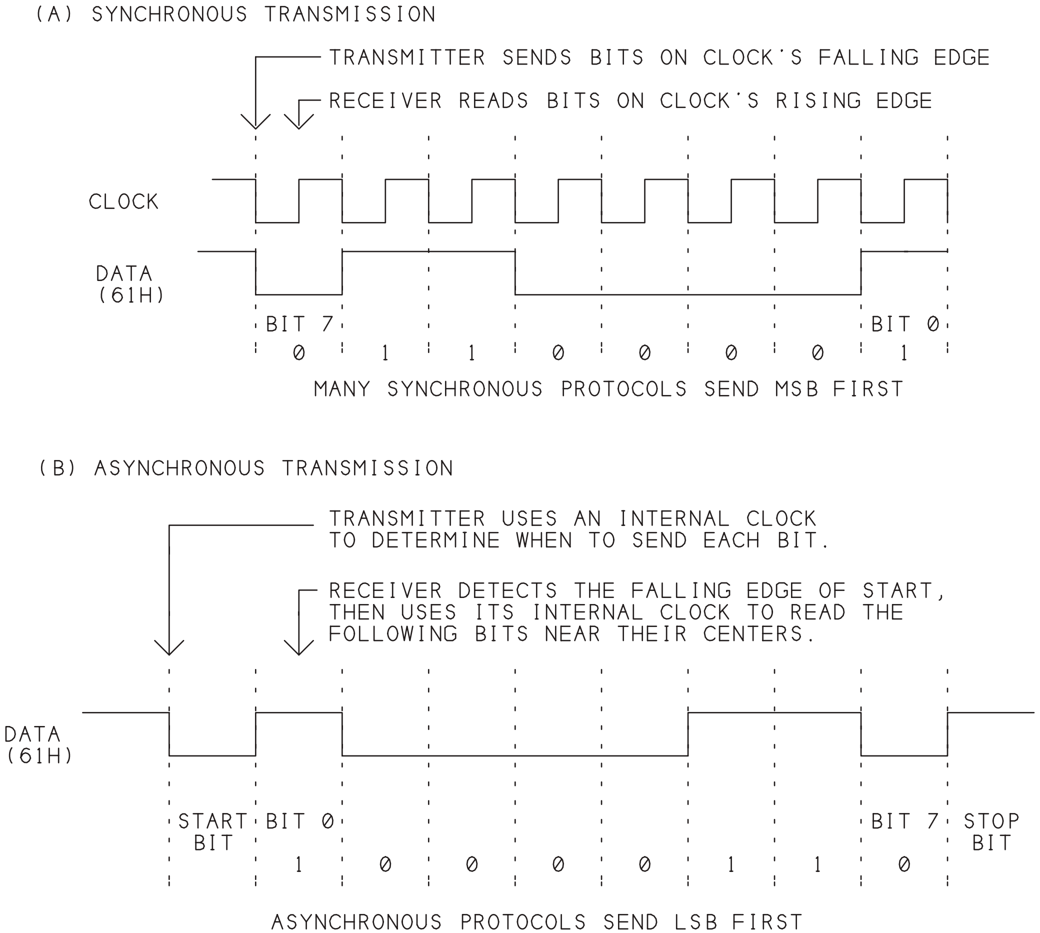


Figure 2-1: Synchronous transmissions include a clock line, while asynchronous transmissions require each computer to have its own clock.

For receivers that require a little extra time to accept received data, some UARTs enable the transmitter to stretch the Stop bit to the width of 1.5 or 2 bits. The original purpose of the longer Stop bit was to allow time for mechanical teletype machines to settle to an idle state.

Bit Rate and Baud Rate

The bit rate is the number of bits per second transmitted or received per unit of time, usually expressed as bits per second (bps). Baud rate is the number of possible events, or data transitions, per second. In the basic, wired, digital transmis-

sions used in this book's examples, each data-transition period represents one data bit, and the bit rate and baud rate are the same. On phone lines, high-speed modems use phase shifts and other techniques to encode multiple bits in each data-transition period, resulting in a baud rate that is less than the bit rate. In popular use, however, the term baud rate often refers to the bit rate.

The number of characters transmitted per second equals the bit rate divided by the number of bits in a word. With 8-N-1 format, a byte of data transmits at 1/10 the bit rate because each word contains 10 bits: 1 Start bit, 8 data bits, and 1 Stop bit. So a 9600-bps link using 8-N-1 format can transmit 960 data bytes per second.

System Support for Low-level Protocols

In PCs and many microcontrollers, a UART handles low-level details of sending and receiving serial data. PC operating systems and programming languages also provide drivers and other support for serial-port communications. Programmers thus can access ports without understanding every detail of the UART's architecture. To gain access to a port, an application selects a bit rate and other port parameters and requests to open, or gain access to, the desired port. To send a byte, the application writes the byte to the transmit buffer of the selected port. The UART then sends the data, bit by bit in the requested format, adding Stop, Start, and parity bits as needed.

In a similar way, the UART stores received bytes in a buffer. After receiving a byte, the UART can generate an interrupt to notify an application of received data, or software can poll the port to find out if data has arrived. The COM-port driver in PCs uses software buffers to supplement the UART's hardware buffers and manages the flow of data between the buffers.

The UARTs in microcontrollers perform the same functions as UARTs in PCs. Some microcontrollers don't have embedded UARTs, or an application might need more serial ports than the hardware provides. In those cases, a system can interface to an external UART or implement a UART in firmware. Parallax, Inc.'s Basic Stamp module is an example of a system with a firmware UART.

Some microcontrollers contain a USART (Universal Synchronous/Asynchronous Receiver/Transmitter), which is similar to a UART but supports both synchronous and asynchronous communications.

Sending Bits

Some knowledge of how serial data transmits is useful in selecting a protocol and interface for a project and in debugging.

The Format

Figure 2-1B shows how a byte transmits in 8-N-1 format. When idle, the transmitter's output is a logic 1. To indicate the beginning of a transmission, the transmitter sends a logic 0 for one bit width. This is the Start bit. At 9600 bps, a bit is 104 µs.

After the Start bit, the transmitter sends the 8 data bits in sequence, beginning with the LSb. The transmitter then sends a logic 1, which functions as the Stop bit. Immediately following the Stop bit or at any time after, the transmitter can send a new Start bit to signify the beginning of a new transmitted word.

At the receiving computer, the transition from logic 1 to the Start bit's logic 0 indicates that a new word is arriving. The transition and the bit rate determine the timing for detecting the bits that follow. The receiver attempts to read the logic state of each bit near the middle of the bit's time period. Reading in the middle of the period helps ensure that the receiver detects the bit values correctly even if the transmitting and receiving clocks don't exactly match in frequency or phase.

RS-232 uses inverted polarities from those shown in Figure 2-1B. An RS-232 Stop bit is a negative voltage and an RS-232 Start bit is a positive voltage. RS-232 interface chips invert the signals and convert to the appropriate voltage levels.

The Need for Accurate Timing

A UART typically uses a receive clock with a frequency 16 times faster than the highest supported bit rate. If the highest bit rate is 9600 bps, the receive clock should be at least 153,600 bps. As Figure 2-2 shows, after detecting the transition that signals a Start bit, the UART waits 16 clock cycles for the Start bit to end, then waits 8 more cycles to read bit zero in the middle of the bit. The UART then reads each bit that follows 16 clock cycles after the previous bit.

If the transmitting and receiving clocks don't match exactly, the receiver will read each new bit closer and closer to an edge of the bit. To read all of the bits in a 10-bit word correctly, the transmit and receive clocks should vary no more than about three percent. With greater variation, by the time the receiver tries

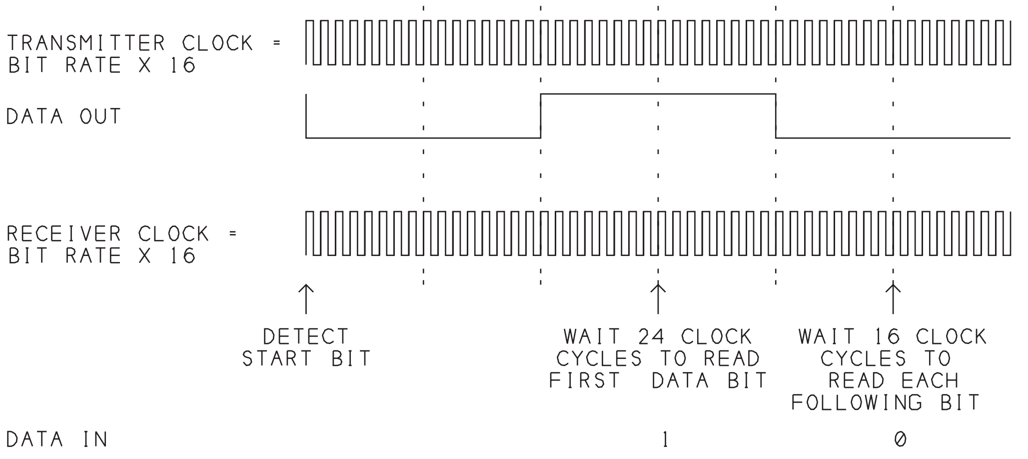


Figure 2-2: To determine when to send data and read received data, the transmitter and receiver each use a clock that is typically 16 times the bit rate.

to read the final bits, the timing may be off by so much that the receiver will read the wrong bits and might not detect the Stop bit. The clocks need to stay in sync only for the length of a word because each word begins with a new Start bit that resynchronizes the clocks.

Because of the need for accurate timing, asynchronous interfaces require stable timing references such as crystal oscillators. For best accuracy, the clock's frequency divided by 16 should be an integer multiple of the highest bit rate the UART supports. For example, the UARTs in early PCs used a 1.8432-MHz crystal. The crystal's frequency divided by 16 equals the common bit rate of 115,200 bps. Newer UARTs often use an 18.432-MHz crystal to enable supporting higher bit rates while allowing compatibility with earlier UARTs.

In a microcontroller, the chip's main timing crystal usually serves as a reference for hardware timers that control the UART's clock. Some microcontrollers, such as Microchip Technology's PIC18F4520, have an internal frequency multiplier that provides a timing reference four times as fast as the external oscillator's frequency.

To reduce errors, many UARTs take three samples in the middle of each bit and use the logic level that matches two or more of the samples. To avoid detecting brief noise glitches as Start bits, some UARTs read the Start bit a second time in the middle of the bit and accept the Start bit only if the bit has remained a logic low.

Autodetecting the Bit Rate

In many serial-port applications on PCs, the user interface provides a combo box where the user can select a bit rate for the port. The ultimate in user convenience is computers that automatically configure themselves to use the same bit rate. To achieve matching bit rates, one computer (we'll call it the sending computer) can detect and match the bit rate of the other computer (we'll call it the receiving computer). There are a couple of ways to implement automatic matching of bit rates.

In one approach, the sending computer establishes communications by repeatedly sending a byte. The byte can be any value from 00h to EFh, where the most significant data bit, which is the last bit to transmit, is zero.

The receiving computer attempts to receive data at the highest supported bit rate. On detecting a Start bit, the computer must receive data for a period equal to or greater than the time required to read a byte at the *lowest* supported bit rate (for example, 33 ms at 300 bps).

If the receiving computer detects more than one received byte, the bit rates don't match. The sending computer tries again using the next lowest bit rate. When the receiving computer detects one and only one received byte, the bit rates match. As an extra check, the receiving computer can verify that the received byte matches an expected value. The receiving computer can send a byte to acknowledge the match, and on receiving the byte, the computers can begin normal communications.

The method works like this: when the receiving computer's bit rate is faster than the transmitting computer's bit rate and the last transmitted data bit is zero, the receiving computer finishes reading the byte while the bits are still arriving. (A receiving computer that doesn't see a logic 1 Stop bit will generate a framing error, but the error is unimportant for detecting the bit rate.) After the receiving computer thinks the word has finished transmitting, any received logic 0 (such as the last bit in the transmitted byte) looks like a Start bit, which causes the receiving computer to try to read another byte. If the receiving computer doesn't see another Start bit, the bit rates match.

Another approach for automatically detecting a bit rate requires code that can measure the width of a a received bit on the port. If the sending computer sends a specific value, the receiving computer can measure a received pulse's width and set a bit rate to match.

The PIC18F4520 contains an enhanced USART that can detect and match a received bit rate. The sending computer must send the value 55h (the character "U"), which results in alternating 1s and 0s in the transmitted word. In auto-baud-rate-detect mode, the microcontroller measures the widths of the incoming bits and sets the serial port's bit rate to the closest match available. Chapter 11 has more about the PIC18F4520.

Autodetecting a COM Port

PCs can have multiple COM ports. To decide which port to use, an application typically provides a combo box that enables the user to select a port. An application can save the name of the last-used port and select that port as a default when available. An application can save other parameters such as bit rate as well. Chapter 9 has more about saving and loading port parameters.

An application can also attempt to find the correct port by sending a defined message to each COM port. The receiving computer watches for the message and sends a reply to let the sending computer know that the correct port has been found. Use this method carefully, however, because unexpected data could cause problems on other COM-port devices.

Data Formats

The data bits in a serial transmission can contain any type of information, including commands, sensor readings, status information, error codes, configuration data, or files that contain text, executable code, or other information. But ultimately all transmitted data is bytes, or units of a different length. (In popular use, a byte is an 8-bit value. Some sources define a *byte* as the smallest number of bits processed by a computer as a unit and use the term *octet* to refer to an 8-bit value. In this book, byte refers to an 8-bit value and unless otherwise specified, the serial port is assumed to be configured for 8 data bits.)

Software that manages serial-port communications typically treats the data being transmitted as either binary or text data.

Binary Data

With binary data, each transmitted byte is a value from 00h to FFh, and the serial-port software assumes nothing about the meaning of the information being transmitted. Higher-level software can interpret and use the data in any way.

The bits in each byte are numbered 0 through 7, with each bit representing the bit's value (0 or 1) multiplied by a power of 2. For example, a byte of 11111111_b translates to FFh, or 255. A byte of 00010001_b translates to 11h, or 17. In asynchronous links, bit zero, the least-significant bit (LSb), arrives first, so if you're looking at the data on an oscilloscope or logic analyzer, remember to reverse the order when translating to a notation with most-significant-bit (MSb) first.

Of course, a serial port can transmit values that have 16, 32, or any number of bits by dividing the value into bytes and transmitting the bytes in sequence. For multi-byte values, the transmitting and receiving computers must agree on the order that the bytes transmit.

Text Data

Some software treats data as text, with each text character expressed as a code. Treating data as text is a convenience for programmers. Most programming languages enable storing text in two ways: as strings, which can contain one or more characters, and as arrays of individual characters. Source code can request to transmit a string such as "hello", and the compiler or interpreter generates code that causes the individual character codes to transmit. Or the source code can request to transmit an array, with each item in the array containing a character code. At the receiving computer, software can store received character codes in a string or array.

For applications that send and receive only basic U.S. English text, encoding usually isn't an issue. When sending or receiving special characters or characters in other alphabets or scripts, the computers must agree on an encoding method.

The .NET Framework and other recent software use encoding methods defined in *The Unicode Standard,* a publication of the Unicode Consortium (*www.unicode.org*). Unicode's encoding methods support over a million characters in dozens of alphabets and other scripts, plus punctuation marks, math and technical symbols, geometric shapes, and other symbols.

Unicode encodes text using code points and code units.

A code point is a value that identifies a character. Unicode code charts assign a code point to each character. The conventional notation for code points is the form U+*code_point*, where *code_point* is a hexadecimal value. For example, the code point for "A" is U+0041. Code points range from U+0000 to U+10FFFF. Each character has one and only one code point.

Software uses the code point to obtain the encoded character, which represents a character using a specific coding method. The code point and encoded character can have the same value or different values depending on the encoding method.

An encoded character that represents a character in software consists of one or more values called code units. A character's code point never changes, but the code unit(s) that make up an encoded character vary with the encoding method. The number of code units that represent a character, their value(s), and the number of bits in the code units vary with the character and encoding method.

The three basic Unicode encoding methods are UTF-8, UTF-16, and UTF-32 (Table 2-1). Each can encode any character that has a defined code point. The encoding methods use different algorithms to convert code points into code units.

UTF-8 encoding uses 8-bit code units, and a UTF-8 encoded character is 1 to 4 code units wide. Basic U.S. English text can use UTF-8 encoding with each character encoded as a single code unit whose value equals the lower byte of the character's code point. The character "A" has a UTF-8 encoding of 41h. The encodings are identical to the ASCII encoding that has been in use for many years. UTF-8 encoding is thus backwards compatible with ASCII encoding.

Basic U.S. English text includes upper- and lower-case Latin letters, the ten digits, and common punctuation. Other values often transmitted are control codes that specify actions such as carriage return (CR), line feed (LF), escape, delete, and so on. The code points for these characters and control codes are in the range U+0000–U+007F. The codes are defined in the Unicode code chart *C0 Controls and Basic Latin*.

For characters with code points of 80h and higher, UTF-8 uses multi-byte encodings of 2 to 4 code units each. If a code unit in a UTF-8 encoded character has bit 7 set to 1, the code unit is part of a multi-byte encoding. UTF-8 thus has no single-byte encoded characters in the range 80h–FFh. Instead, characters with code points in this range use encodings with multiple code units. For example, the © character has a code point of A9h and a 2-byte UTF-8 encoding of C2h A9h.

The chart that defines code points U+0080–U+00FF is *C1 Controls and Latin-1 Supplement*. Many of these code points are assigned to accented characters for European languages and additional control codes.

Table 2-1: Unicode encoded characters can use any of three encoding methods.

Encoding Method	Bits per Code Unit	Code Units per Character
UTF-8	8	1, 2, 3, or 4
UTF-16	16	1 or 2
UTF-32	32	1

ANSI encoding is a legacy encoding method usually defined as the text and control codes encoded according to a draft of an ANSI standard that Microsoft implemented as code page 1252. (A code page is a table that defines character encodings for a specific language.) UTF-8 is not backwards compatible with ANSI encoding, which uses single-byte values in the range 80h–FFh. For example, the ANSI encoding for © is A9h, but UTF-8 uses a 2-byte encoding for this character.

UTF-16 encoding uses 16-bit code units, and UTF-16 encoded characters are 1 or 2 code units each. UTF-16 encoding represents more than 60,000 characters as single code units whose values equal the characters' code points. For example, "A" is 0041h, and © is 00A9h. Characters with code points greater than FFFFh are encoded as a pair of code units called a surrogate pair.

UTF-32 encoding uses 32-bit code units. A UTF-32 encoded character is always a single code unit. A UTF-32 code unit always has the same value as the character's code point. For example, "A" is 00000041h, and © is 000000A9h.

The UTF-16 and UTF-32 methods have alternate forms to enable storing code units as big endian (storing the most significant byte first in memory) or little endian (storing the least significant byte first in memory). The unmarked forms (UTF-16, UTF-32) are big endian unless the data is preceded by a byte-order mark (FEFFh). On seeing a byte-order mark of FFFEh, the receiving computer should reverse the byte order in the code units that follow. On seeing a byte-order mark of FEFFh, the receiving computer should *not* reverse the byte order in the code units that follow. The byte-order mark is the only defined use for values FEFFh and FFFEh; the values don't appear in any character encodings.

The BE forms of the encoding methods (UTF-16BE, UTF-32BE) are always big endian. The LE forms (UTF-16LE, UTF-32LE) are always little endian. Again, computers can use any encoding method as long as both computers understand what encoding the other computer is using.

The Unicode standard has details about how to convert code points into code units using each encoding method. A .NET application can use methods that perform the encoding and decoding automatically. Chapter 9 has more about working with text in .NET applications.

ASCII Hex

Treating data as text is the obvious choice for transferring strings or files that contain text. But you can also use text to transfer binary data by expressing the data in ASCII Hex format. Each byte is represented by a pair of ASCII codes that represent the byte's two hexadecimal characters, which are in the range 0–9 and A–F. ASCII Hex can represent any numeric value using only the ASCII codes 30h–39h (to represent values 00h–09h) and 41h–46h (to represent values 0Ah–0Fh). Code that allows lower-case letters might also use 61h–66h (to represent 0ah–0fh).

Instead of sending one byte to represent a value from 0 to 255, the transmitting computer sends two bytes, one for each character in the hex number that represents the byte. The receiving computer can convert the characters to numeric values or use the data in any way.

For example, consider the decimal number:

```
225
```

Expressed as a binary number, it's:

```
11100001
```

In hexadecimal, it's:

```
E1
```

The ASCII codes for "E" and "1" are:

```
45h 31h
```

So the binary representation of this value in ASCII hex consists of these two bytes:

```
01000101  00110001
```

A serial link using ASCII Hex format would send the decimal value 225 by transmitting the two bytes above.

A disadvantage of using ASCII hex is that each byte value requires two characters so data takes twice as long to transfer. Also, in most cases the application at each end must convert between ASCII hex and binary.

Still, ASCII Hex has benefits. One reason to use ASCII Hex is to free all of the other codes for other uses, such as flow-control codes, an end-of-file indicator, or network addresses. ASCII Hex also allows protocols that support only seven data bits to transmit any numeric value.

Other options are to send values as ASCII decimal, using only the codes for 0 through 9, or ASCII binary, using just 0 and 1.

Here are functions that convert bytes to and from ASCII Hex format:

VB

```
Private Function ConvertAsciiHexToByte(ByVal asciiHexToConvert As String) As Byte

        Dim convertedValue As Byte
        convertedValue = Convert.ToByte(asciiHexToConvert, 16)
        Return convertedValue

End Function

Private Function ConvertByteToAsciiHex(ByVal byteToConvert As Byte) As String

        Dim convertedValue As String
        convertedValue = Hex$(byteToConvert)
        Return convertedValue

End Function
```

VC#

```
private byte ConvertAsciiHexToByte( string asciiHexToConvert )
{
        byte convertedValue;
        convertedValue = Convert.ToByte( asciiHexToConvert, 16 );
        return convertedValue;
}

private string ConvertByteToAsciiHex( byte byteToConvert )
{
        string convertedValue = null;
        convertedValue = System.Convert.ToString
                (System.Convert.ToByte(byteToConvert), 16).ToUpper();
        return convertedValue;
}
```

Application-specific Protocols

An application-specific protocol can define the contents of the data being transmitted and might also specify lower-level parameters such bit rate and number of data bits. The protocol can be a defined industry standard or a vendor-specific protocol. Two examples of industry-standard protocols are the "AT" commands and related protocols for modems and the NMEA 0183 protocol for global positioning systems.

Modems

One way computers access the outside world is via modems that connect to the phone system. A phone modem converts digital data transmitted by a computer to an analog signal for transmitting on the phone line. In the other direction, the modem convert's the phone line's analog signal to digital data to send to the computer. Older modems connected to RS-232 serial ports on PCs, while recent modems typically reside on the system bus. Embedded systems can incorporate modems or access external modems.

For communicating with computers, many modems support a protocol descended from the "AT" command set and related protocols first defined by modem manufacturer Hayes Microcomputer Products, Inc. A specification that documents the commands and protocols is V.250 from the International Telecommunication Union (*www.itu.int*).

The AT commands are plain text. Most begin with the letters "AT". A computer can use the commands to communicate with a modem. For example, in this command:

```
ATDT5552468
```

"AT" begins the command, "D" means dial, "T" means use touch tones to dial, and "5552468" is the number to dial. The command ends in a CR.

This command requests the modem to echo everything the modem transmits back to the transmitting computer:

```
ATE1
```

This command turns echoing off:

```
ATE0
```

Modems that support AT commands have two operating modes. In command mode, the modem responds to received commands. In data mode, the modem transmits and receives data over the phone lines. To switch from data mode to command mode, a computer sends an escape sequence. The sequence is typi-

cally "+++" sent with specific timing requirements to minimize the chance of switching in error if transmitted data happens to contain the sequence. The modem responds with "OK". Switching from command mode to data mode is automatic after some commands, or a computer can send the "ATO" command to request a switch.

Global Positioning Systems (GPS)

Many global positioning system (GPS) devices have either an RS-232 port or a USB port that functions as a virtual COM port. These devices typically communicate with PCs using protocols defined in the NMEA 0183 Interface Standard available from *www.nmea.org*.

The standard specifies parameters for serial communications and a format for transmitted data. The communication parameters are typically 4800 bps, 8-N-1. (A high-speed addendum to the standard supports communications at 38,400 bps, and some devices support additional bit rates. Devices with USB virtual COM ports don't use the serial parameters.)

The data is plain text, 8 bits per character. The devices transmit the data in blocks called sentences. Standard sentences use this format:

Number of Bytes	Description
1	Initial character. Always "$".
2	Talker ID. Identifies the device type.
3	Sentence ID. Identifies the type of information being sent.
variable	Data, comma delimited.
3	"*" character followed by a 2-byte checksum. Optional for some sentences.
2	End-of-sentence indicator. Always CR LF.

In addition, vendors can define proprietary sentences, which begin with "$P" followed by a 3-letter manufacturer ID, vendor-specific data, and CR and LF codes to end the command. For GPS example code, see Richard Grier's Hard & Software (*www.hardandsoftware.com*).

Preventing Missed Data

Most computers have other things to do besides waiting to receive data on a serial port. A data-acquisition system might collect and store data to send at

intervals to a remote computer. Or a device might be responsible for monitoring and controlling equipment while occasionally sending information or receiving instructions via a serial link.

A transmitting computer might want to send data at a time when the receiving computer is occupied with something else. Hardware and programming can help ensure that a receiver sees all of the transmitted data and that data arrives without errors.

Ways to ensure that data arrives without errors include flow control, buffering, use of polling or interrupts to detect received data, error checking, and acknowledging received data. The descriptions below are an introduction to these concepts. The chapters that follow show how to implement the concepts in applications.

Flow Control

With flow control, a transmitting computer can indicate when it has data to send, and a receiving computer can indicate when it's ready to receive data. The computers in a serial link should use flow control unless the receive buffers are large enough to hold all of the data that might arrive before the receiving computer can read the data from the buffer.

In a common form of hardware flow control, the receiver sets a dedicated line to a defined state when ready to receive data. The transmitting computer checks the state of the line before sending data. If the line isn't in the expected state, the transmitting computer waits. Flow control in both directions requires a line for each direction. Flow control is sometimes called handshaking. However, a full handshake requires 2-way communication: the transmitting computer indicates that it has data to send, and the receiving computer indicates when it is ready to receive the data.

The RS-232 specification assigns names to flow-control signals. On a PC, the input signal is Clear To Send (CTS) and the output signal is Request to Send (RTS). (The names reflect an alternate usage of the lines to implement a full handshake.) A cable that connects two PCs must connect each RTS output to the other computer's CTS input. A positive RS-232 voltage means ready to receive and a negative voltage means not ready.

Microcontrollers typically don't have dedicated CTS and RTS lines. Device firmware can use any spare port pins for flow control. Source code can use the names RTS and CTS or use names such as flow_control_in and

flow_control_out to eliminate confusion about which signal is the input and which is the output.

Two additional RS-232 flow-control signals are Data Terminal Ready (DTR) and Data Set Ready (DSR). These lines were defined as a means for providing information about the status of a phone line or other communication channel on a modem that connects via RS-232 to a computer or terminal. On PCs, DTR is an output and DSR is an input. Microcontrollers typically don't have dedicated DTR and DSR lines. Spare port pins with firmware support can provide these signals when needed. Chapter 5 has more about RS-232 signals.

Some links can use software flow control, where a receiving computer sends an Xon code to indicate that the computer is ready to receive and sends an Xoff code to tell the transmitter to stop sending. This method works only when sending data such as plain English text or another encoding that doesn't use the Xon and Xoff codes in other data. The Xon code point is typically 11h (Control+Q), and Xoff is 13h (Control+S). Some software drivers enable selecting different codes.

A link can use hardware and software flow-control methods at the same time. The transmitting computer sends data only if the remote computer's CTS line is high and the transmitting computer hasn't received an Xoff.

Buffers

Hardware and software buffers can help to prevent missed data and enable data to transfer as quickly as possible. The buffers can store received data and data waiting to be sent. On a port without flow control, a receive buffer can prevent missed data by storing received data until program code can retrieve the data. On a port with flow control and a receive buffer, the transmitting computer can send large quantities of data even if the receiving computer can't process the data right away. Transmit buffers can enable software to store data to be sent and move on to other tasks.

The buffers can be in hardware, software, or both. Serial ports on PCs typically have 16-byte hardware buffers built into the UARTs. In the receive direction, the UART can store up to 16 bytes before software needs to read them. In the transmit direction, the UART can store up to 16 bytes and transmits the bytes using the selected protocol. Some UARTs, including those in many Virtual COM-port devices, have larger hardware buffers.

COM-port drivers in PCs maintain software buffers that are programmable in size and can be as large as system memory permits. The driver transfers data between the software and hardware buffers as needed.

Buffers in microcontrollers tend to be small. Some UARTs have no hardware buffers at all. A computer with small or non-existent buffers may need to use other methods to prevent lost data.

Event-driven Programming and Polling

Events that can occur at a serial port include sending and receiving data, changes in flow-control signals, errors in received data, and timeouts when attempting to send or receive data. There are two ways for program code to detect these events.

One way is to jump to a routine when an event occurs. The code responds quickly to port activity without having to waste time checking only to learn that no activity has occurred. This type of programming is called event driven because an external event can break in at any time and cause the program's execution to branch to a routine to handle the event.

The .NET Framework's SerialPort class includes the DataReceived, PinChanged, and ErrorReceived events. Handler routines can execute when a software buffer's count reaches a trigger value, when the state of a flow-control or status pin changes, and when an error or timeout occurs. Many microcontrollers have hardware interrupts that can perform similar functions.

The other approach to detecting events is to poll the port by periodically reading properties, signal states, or registers to find out if an event has occurred. This type of programming doesn't use a port's hardware interrupts. The code has to poll often enough to detect all data and events in time to prevent lost dat or other problems. The needed frequency of polling depends on buffer size, the amount of data expected, and whether the code must respond quickly to events. For example, a device that has a 16-byte buffer and polls the port once per second can receive no more than 16 bytes per second or the buffer might overflow and data will be lost.

Polling is often appropriate when transferring short bursts of data or when a computer sends a command and expects an immediate reply. A polled interface doesn't require a hardware interrupt, so you can use this type of programming on a port that doesn't support hardware interrupts. The code can perform the polling in a task loop that repeatedly performs required tasks, or a timer interrupt can schedule tasks at intervals.

Acknowledgments

In some applications, the transmitting computer needs an acknowledgment that the data was received. Acknowledgments are especially useful in networks where multiple computers share a communications path and a driver's switching on at the wrong time can block another computer's transmission.

An acknowledgment can be a defined value, or the transmitting computer can assume that a computer received its message on receiving a response with requested data or other information. A transmitting computer that doesn't receive an expected response can assume there is a problem and retry or take other action.

When sending to a computer that has no input buffer or a very small buffer, a transmitting computer can use a full handshake to ensure that the receiving computer is ready to receive a block of data. The transmitting computer can begin by sending a code, repeatedly if needed, to announce that the computer wants to send data. On detecting the code, the receiving computer can send a code to acknowledge and then devote full attention to monitoring the serial input. On seeing the acknowledgment, the transmitting computer knows it's OK to send the rest of the data.

Error Checking

A receiver can use error checking to verify that all data arrived correctly. Ways to check a message for errors include parity bits, checksums, and sending duplicate data.

As described earlier in this chapter, a parity bit in each transmitted word enables the receiving computer to detect most errors introduced between the transmitter and receiver. When using .NET's SerialPort class or other serial-port classes or libraries for PC applications, the application only needs to select a parity type. The software automatically calculates and places the correct parity bit in each transmitted word and can raise an error on receiving data with incorrect parity. Microcontroller hardware and software may require the firmware to calculate and set or check the parity bit for each transmitted and received word.

A checksum is an error-checking value obtained by performing mathematical or logical operations on the contents of a block of data. Applications can choose from a variety of methods to calculate checksums.

A basic checksum calculation adds the values of the bytes in a block and uses the lowest byte of the result as the checksum. A checksum for ASCII Hex data

can add the values represented by each pair of characters. Intel Hex and Motorola S-Record are two data formats that uses checksums on ASCII Hex data.

The cyclic redundancy check (CRC) method uses more complex calculations to obtain checksum values. Protocols that use CRC values include the file-transfer protocols Kermit, XModem, YModem, and ZModem.

Hash values are very secure checksums produced by message detection code (MDC) hash functions. To use hash values, the sender and receiver must share a key, which is a value used in creating the hash value and in verifying the received data.

A computer that receives data with a checksum can repeat the calculation to obtain the checksum. If the checksum doesn't match the expected value, the computer knows it didn't receive the same data the transmitting computer sent. A computer that detects an error can notify the sending computer so it can try again or take other action. After a number of unsuccessful tries, the transmitting computer can give up, display an error message, or sound an alarm as needed. A checksum adds little overhead to large data blocks.

A receiving computer should also know what to do if a message is shorter than expected or if expected data or an end-of-message code doesn't arrive. Instead of waiting forever, the software should eventually time out and can attempt to notify the sending computer if needed.

In another form of error checking, the transmitter sends each message twice and the receiver verifies that the message is the same both times. Of course this means each message takes twice as long to transmit. Sending duplicate data can be useful when sending occasional, short bursts of data in an environment prone to errors. Many infrared data links use this method.

3

COM Ports on PCs

This chapter explores the options for COM ports on PCs, including ports in USB/serial converters and ports in serial servers on networks.

Port Architecture

As Chapter 1 explained, COM ports on PCs can include ports on motherboards, expansion cards, USB converters, and serial servers. Other names for COM ports are communications port and Comm port.

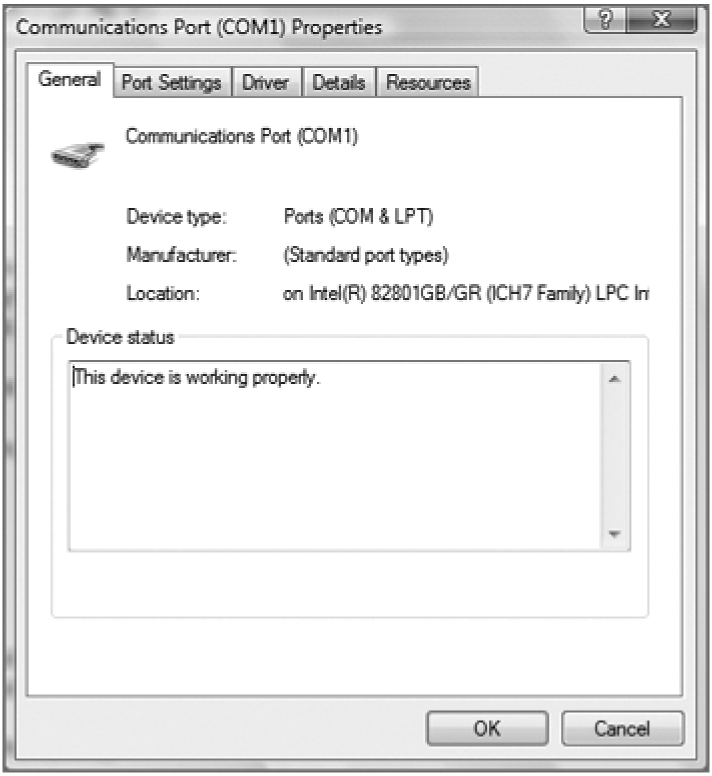
For each COM port, an operating-system driver assigns a symbolic link name such as COM1, COM2, and so on, which applications use to detect and access the port. Recent Windows editions don't limit the number of COM ports. Of course, every system has finite resources that limit how many COM ports can be in use at the same time.

Device Manager

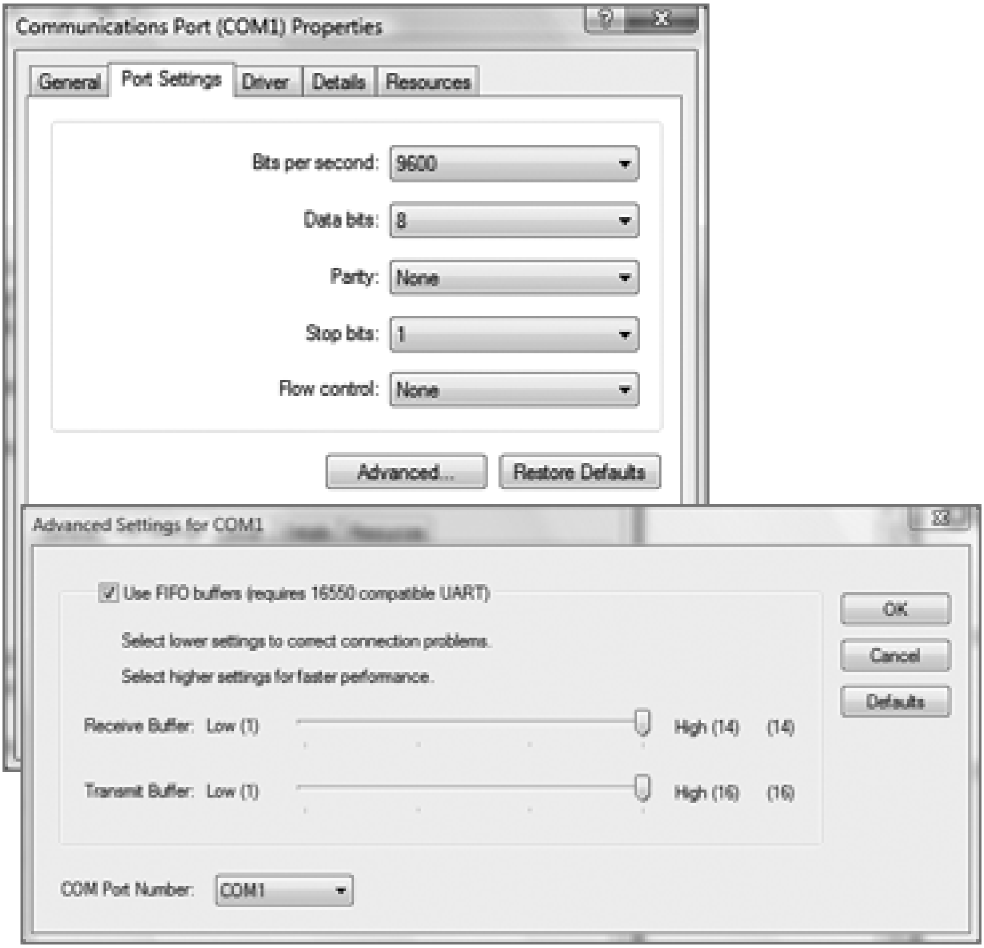
The Windows Device Manager shows information about each COM port. To access the Device Manager, right click on My Computer, click Manage, and in the Computer Management pane, select Device Manager. Or click Start and

select Settings > Control Panel > System > Hardware > Device Manager. Or save some clicks by creating a shortcut to the file *devmgmt.msc* in *Windows\System32*.

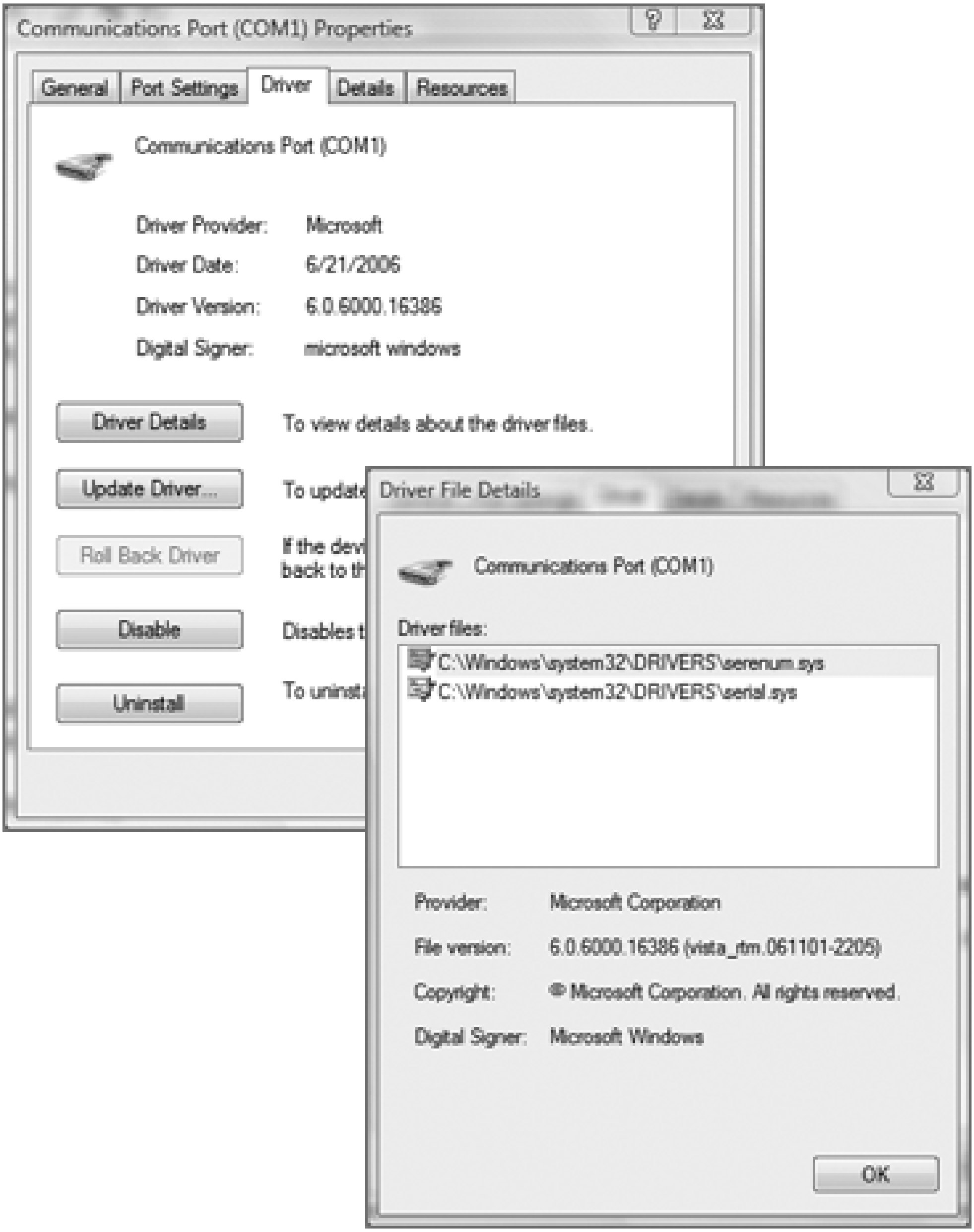
To view a COM port in the Device Manager, click Ports (COM & LPT), right-click a COM port, and select Properties. The Properties window has several tabs that display the port's property pages. A vendor-provided co-installer can supply custom property pages for vendor-specific device properties. The pages shown below are typical.



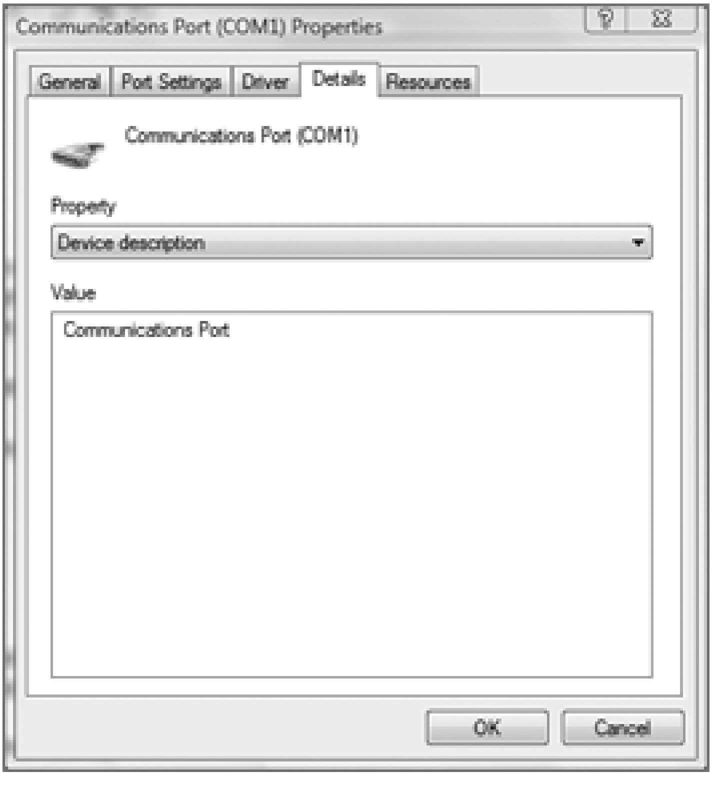
The General tab has basic information about the port.



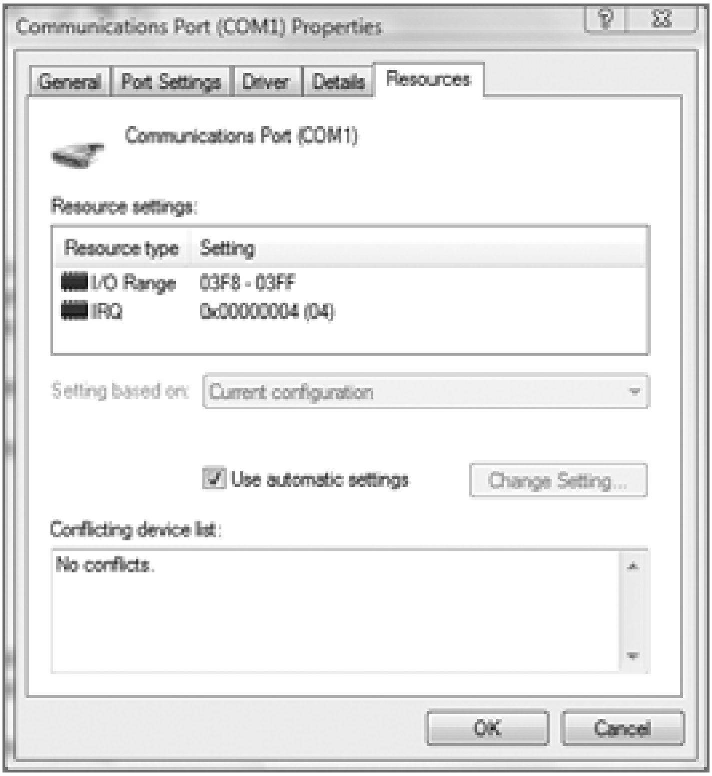
The Port Settings tab displays the default settings for the port. A USB/serial converter doesn't use the bit rate, parity, and Stop bits in communications between the PC and the converter, but a driver can send these values to a device that uses the settings on its serial port. For example, a USB/RS-232 converter can set the parameters of its RS-232 port to match values specified on the PC. The Advanced Settings window enables setting buffer properties and the COM-port number. Applications can change the parameters from the default values set in the Device Manager.



The Driver tab enables updating, disabling, and uninstalling the drivers assigned to the port. The Driver Details window lists the drivers the device uses.



The Details tab has a drop-down Property box that provides access to a variety of information about a port, including device instance, hardware, and compatible IDs, class installers and co-installers, and power-state information. In the window shown, the Value pane shows the "Device description" property.



Ports on motherboards or expansion cards have a Resources tab that displays the port addresses and IRQ line the port uses. If you uncheck *Use automatic settings,* you may be able to change the resources.

Port Resources

A typical COM port on a motherboard or expansion card contains a UART that interfaces to the system bus, typically a PCI bus. Each UART uses a series of eight port addresses. The first address in the series is the port's base address. Most of these ports also have an assigned IRQ line to carry interrupt requests.

A port can use any addresses and IRQ lines supported by the system hardware. These are the addresses and IRQ lines allocated to COM ports in early PCs:

Port	Address	IRQ
COM1	3F8h	4
COM2	2F8h	3
COM3	3E8h	4 or 11
COM4	2E8h	3 or 10

Older ports often have jumpers, switches, or configuration utilities that enable selecting a base address and IRQ line. The setup screens that you can access on boot up can enable configuring motherboard ports. With some hardware, multiple ports can share an IRQ line.

A port on a USB/serial converter doesn't have its own addresses and IRQ line. Instead, the port uses the shared resources of the Universal Serial Bus.

Serial Servers

A serial server is a device that enables accessing serial ports over a network. A typical serial server contains a microcontroller, an Ethernet controller, and one or more UARTs that interface to RS-232 or RS-485 ports. The server manages communications between the Ethernet network and the serial ports. A serial server can also interface to a wireless (Wi-Fi) network. Many sources offer serial-server modules.

Serial servers can use defined Internet protocols for network communications. The specifications are available from *www.rfc-editor.org*.

Most serial servers communicate via TCP (specification STD0007), which defines a way of establishing a connection to a device and exchanging data with acknowledgements, sequence numbers, and other features that help ensure reliable transfers. Each serial port uses a separate TCP connection. UDP (STD0006) is an alternative for applications that don't require TCP's reliability. Some serial servers use Telnet connections. The Telnet specification (STD0008) defines a protocol for transmitting characters and control data over a TCP connection.

Information specific to COM ports and modems can use protocols defined in *Telnet Com Port Control Option* (RFC 2217). This document defines commands for setting COM-port parameters and flow-control methods, reading

RS-232 status signals, writing to RS-232 control signals, and reading error information and other line-state data.

On a PC, COM-port redirector software can cause a serial server's serial ports to appear as local COM ports. Application software can then access the remote ports as if they were local ports.

Accessing Ports

Low-level drivers on the PC handle hardware-specific details such as detecting ports, assigning names, and managing communications with applications.

Drivers

Several drivers provided with Windows each manage different aspects of COM-port communications. The *serial.sys* driver controls communications with COM-port devices. The *serenum.sys* driver is an upper-level filter driver that enumerates and retrieves identifying information from Plug-and-Play COM-port devices

A USB device accessed as a COM-port device requires an additional driver. The *usbser.sys* driver is a bus driver that manages communications between the operating system's COM-port driver and USB drivers. The *usbser.sys* driver requests to send and receive COM-port data in USB bulk transfers and implements protocols for sending and receiving status and control information in USB transfers. Some USB virtual COM-port devices use vendor-specific drivers in place of the Windows-provided drivers. Chapter 15 and Chapter 16 have more about USB virtual COM ports.

Many devices can use the Windows-provided drivers or a driver provided by a USB-controller manufacturer or another third party. For low-level programming, Microsoft's Windows Driver Kit (WDK) provides a variety of example code, including source code for the *serial* and *serenum* drivers. The *pnpports* example demonstrates a class installer and property page provider for the Ports class. The *devcon* example shows how to obtain information about installed devices and how to perform functions such as enabling, disabling, restarting, updating drivers for, and removing devices.

Identifying Ports

Windows uses several methods to detect COM ports:

For internal Plug-and-Play ports, the *serenum.sys* driver retrieves a PnP ID from the port. The PnP ID contains manufacturer and product identifiers and other optional information. Microsoft's *Plug and Play External COM Device Specification* defines the protocol for retrieving the PnP ID.

For legacy (non Plug-and-Play) ports, the *serial.sys* driver reads port settings from the system registry. The settings are in a COM port subkey under *..\Services\Serial\Parameters*.

For USB virtual COM-port devices, a USB driver requests a series of data structures called descriptors from the device. The descriptors contain information that identifies an INF file. The INF file names the device's driver, which determines how application access the device.

For COM ports on serial servers, COM-port redirector software creates and installs virtual COM ports on the PC.

GUIDs for COM Ports

COM-port software can use two Globally Unique Identifiers (GUIDs) to find and gain access to devices. A GUID is a 16-byte value that can identify devices that behave in similar ways.

Device Setup GUID

A device setup GUID identifies devices that have similar capabilities and are installed and configured in the same way. Devices that have the same device setup GUID use the same class installer to assign drivers. COM-port devices are in the Ports class. The class's GUID is GUID_DEVCLASS_PORTS defined in *devguid.h* in the WDK. The class also includes parallel (LPT) ports.

COM-port devices in the Ports class must support the software interface defined in *ntddser.h* in the WDK. The file defines functions compatible with the 16550 UART chip used in many early serial ports. Devices aren't required to implement the 16550's asynchronous serial interface, however. For example, a USB virtual COM port in the Ports class might have a parallel or other interface.

Some intelligent multiport serial cards instead use the MultiportSerial device setup class (GUID_DEVCLASS_MULTIPORTSERIAL).

Device Interface GUID

A device interface GUID identifies devices that applications can access in the same way. During device installation, a driver registers the device as an instance of a device interface class and associates a symbolic link name with the device.

The device interface GUID used by internal serial ports is GUID_DEVINTERFACE_COMPORT defined in *ntddser.h*. (Some older software uses GUID_CLASS_COMPORT instead.) The symbolic link name has the form COM<*n*>, where *n* is a unique number.

Applications can use SetupDi API functions to retrieve the symbolic link names of all ports whose drivers have registered the port using GUID_DEVINTERFACE_COMPORT. Applications that use the .NET Framework and other programming tools typically use higher-level functions to find ports and thus don't need to use the GUID directly.

COM Port Numbering

Windows provides a COM-port database that helps ensure a unique number for each port. Application programming interface (API) functions provide a way for applications to claim and release port numbers. (See the *pnpports* example in the WDK for example code.) Uninstalling a port from within the Device Manager releases the port number for use by other port hardware.

To prevent unwanted "COM-port proliferation," include a serial number in a string descriptor in any USB virtual COM port you design. A device that contains a serial number retains its COM-port number even if moved to a different USB port on the system. A device that doesn't contain a serial number gets a new port number on each attachment to a different port on the system. (Two devices with identical USB descriptors and no serial numbers will get the same COM-port number if they attach to the same port at different times.) Chapter 16 has more about storing serial numbers in USB devices.

INF Files

Windows uses INF files to match a device setup class to a device. Devices in the Ports class use the INF file *msports.inf* (in *WINDOWS\inf*), which contains the device setup GUID for the Ports class. A USB virtual COM-port device must also have a device-specific INF file as explained in Chapter 16.

Options for Application Programming

For .NET's programmers, the SerialPort class provides a convenient way to access COM-port devices. Other programming options include API functions such as CreateFile, ReadFile, and WriteFile and software components from Greenleaf Software, MarshallSoft Computing, Sax.net, and others. Chapter 9 and Chapter 10 have more about .NET programming.

Terminal Emulators

Terminal-emulator utilities can provide a user interface for COM-port communications. With a typical terminal emulator, you can type data to send, view received data, and transfer files. Hilgraeve, Inc. (*www.hilgraeve.com*) provides a free edition of the Hyperterminal utility for personal use as well as a business edition of Hyperterminal and a more full-featured emulator called HyperACCESS. Windows editions through Windows XP include Hyperterminal, accessed via *Start* > *Programs* > *Accessories* > *Communications*. Other terminal emulators are available from a variety of sources. A popular free emulator is Tera Term Pro.

Direct Port Access

PC programmers sometimes want to know how to access a UART's registers directly. While direct port access is possible for some ports, it's rarely useful.

Under DOS and early Windows editions, most serial ports used standard port addresses, and applications could read and write to these addresses using assembly code or functions, often called Inp and Out.

Under recent operating systems, accessing port addresses requires a low-level driver such as inpout32 (available from *www.Lvr.com*). USB virtual COM port devices don't have dedicated port addresses and might not have a UART at all. Beyond these limitations, using .NET's SerialPort class or an equivalent driver has many benefits. Applications can still access the UART's registers, but do so indirectly and without having to know the port address or other hardware details. Plus, serial-port classes and drivers offer built-in efficiencies such as buffering of serial data.

This page intentionally left blank

4

Inside RS-232

RS-232 is an interface that is suitable for many basic communication tasks between two computers. This chapter introduces RS-232 signals and interfacing options.

The Hardware Interface

RS-232 is designed to handle communications between two devices with a distance limit of around 80 to 130 ft, depending on the bit rate and cable type. RS-232 uses unbalanced, or single-ended, lines. Each signal in the interface has one dedicated line whose voltage is referenced to a common ground.

Signals

In popular use, RS-232 refers to a serial interface that complies with much of the standard *TIA-232-F: Interface between Data Terminal Equipment*. The name RS-232 dates to an earlier edition of the standard.

The standard's current publisher is the Telecommunications Industry Association (TIA). Previous versions were a product of the Electronics Industries Association (EIA). A similar standard is encompassed by V.24 and V.28 from the

International Telecommunication Union (ITU) and ISO 2110 from the International Organization for Standardization (ISO).

The standard defines the names and functions of signals, electrical characteristics of the signals, and mechanical specifications, including pin assignments. Earlier versions didn't include all of these items. The addition of new material, such as recommended connectors, documented what had become standard through popular use.

The standard designates 25 lines in the interface, but RS-232 ports rarely support more than the nine signals in Table 4-1. The additional signals are intended for use with synchronous modems, secondary transmission channels, and selecting a transmission speed on a dual-rate modem. Some applications require only three lines (or even two, if the link is one way).

Much of the RS-232 terminology reflects its origin as a standard for communications between a computer terminal and an external modem. A "dumb" terminal contains a keyboard, a display, a communications port for accessing a remote computer, and little else. An RS-232 link connects the terminal to a modem, which in turn accesses the phone lines that connect to the remote computer. PCs with modems and network interfaces have made this type of terminal connection nearly obsolete.

These days, an RS-232 port is more likely to connect a PC to an embedded system or to connect two embedded systems. Much of the original RS-232 terminology thus doesn't apply to modern applications, but the hardware interface remains useful.

DTE and DCE

The RS-232 standard calls the terminal end of the link the data terminal equipment, or DTE. The modem end of the link is the data circuit-terminating equipment, or DCE.

The signals and their functions are named from the perspective of the DTE. For example, TX (transmit data) is an output on a DTE and an input on a DCE, while RX (receive data) is an input on a DTE and an output on a DCE.

The RS-232 ports on PCs are almost always DTEs. It doesn't matter which device in a link is the DTE and which is the DCE, but every connection between two computers must either have one of each or must emulate the absent interface (typically DCE) with an adapter called a null modem. The null modem swaps the lines so each output connects to its corresponding input. Chapter 5 has more about these adapters.

Table 4-1: The PC's serial port and many other interfaces use at most the nine pins named here.

Pin Number (9-pin D-sub)	Pin Number (25-pin D-sub)	Signal	Source	Type	Description
1	8	CD	DCE	control	Carrier detect
2	3	RX	DCE	data	Receive data
3	2	TX	DTE	data	Transmit data
4	20	DTR	DTE	control	Data terminal ready
5	7	SG	–	–	Signal ground
6	6	DSR	DCE	control	Data set ready
7	4	RTS	DTE	control	Request to send
8	5	CTS	DCE	control	Clear to send
9	22	RI	DCE	control	Ring Indicator
–	1, 9-19, 21, 23-25	unused	–	–	–

The Nine Lines

These are the three essential lines for 2-way RS-232 communications:

TX. Carries data from the DTE to the DCE. Sometimes called TD or TXD.

RX. Carries data from the DCE to the DTE. Sometimes called RD or RXD.

SG. Signal ground. Sometimes called GND or SGND.

The remaining lines are flow-control and other status and control signals. The RS-232 standard defines uses for all of the signals, but applications are free to use the signals in any way as long as both ends agree on what the signals mean.

Many links use the RTS and CTS flow-control signals. As Chapter 2 explained, in the most commonly used protocol, each computer uses an output bit to let the other computer know when it's OK to send data. The DCE asserts CTS when ready to receive data, and the DTE asserts RTS when ready to receive data. Before transmitting, a computer reads the opposite computer's flow-control output. If the signal's state indicates that the receiving computer isn't ready for data, the transmitting computer waits.

In links that use DTR and DSR, each computer typically asserts its output on power up to indicate that the equipment is present and powered.

Software drivers for some USB virtual COM ports don't allow applications to read CTS directly. An application that requires direct reading of an input bit can use DSR instead.

RI and CD are outputs on the DCE and can have any use if not used for their original purposes of detecting a ring signal on a phone line (RI) and detecting the presence of a carrier (CD).

All RS-232 inputs and outputs must be able to withstand a short circuit to any other RS-232 signal, including ground, without damage.

Break Signaling

The Break signal supports a rarely used form of signaling. Setting the break signal in a UART causes TX to remain at a positive RS-232 voltage, which is the opposite of the idle-state voltage.

The break signal enables in-line signaling to a microcontroller or other device that has no input buffers or flow-control lines. The signal also provides a way to toggle the TX line as desired for any purpose.

Shield Ground

In the 25-pin connector, pin 1 is a shield connection to allow grounding of a cable shield. The pin isn't always connected inside the device, and the 9-pin connector doesn't include the pin. Many shields instead use the connector shell to connect to a grounded chassis. The shield normally connects to the chassis, r frame, of the DTE only, not the DCE.

Earlier versions of the standard called pin 1 protective ground, and the wire sometimes connected the chassis of the equipment on both ends. Each chassis in turn connected to a safety ground at the equipment's power plug. TIA-232 recommends using a separate wire (not one of the wires in the connector) to connect the grounds of the two chassis if needed.

Voltages

RS-232 logic levels are defined as positive and negative voltages rather than the positive-only voltages of TTL and CMOS logic (Table 4-2). At an RS-232 data output (TX), logic 0 is defined as equal to or more positive than +5V, and logic 1 is defined as equal to or more negative than -5V. In other words, the data uses negative logic, where the more positive voltage is logic 0 and the more negative voltage is logic 1.

Table 4-2: RS-232 uses positive and negative voltages.

Parameter	Voltage (V)
Logic 0 or On output	+5 to +15
Logic 1 or Off output	-5 to -15
Logic 0 or On input	+3 to +15V
Logic 1 of Off input	-3 to -15

The status and control signals use the same voltages, but with positive logic. A positive voltage indicates a logic 1 and a function that is On, asserted, or True, and a negative voltage indicates a logic 0 and a function that is Off, not asserted, or False.

An RS-232 interface chip typically inverts the signals and converts between TTL/CMOS voltages and RS-232 voltages. On a UART's output pin, a logic-1 data bit or an Off control signal is a logic high, which results in a negative voltage at the RS-232 output. A logic-0 data bit or asserted control signal is a logic low at the UART and results in a positive voltage at the RS-232 output.

Because an RS-232 receiver can be at the end of a long cable, by the time the signal reaches the receiver, the voltage may have attenuated or have noise riding on it. To allow for degraded signals, the receiver accepts smaller voltages as valid. A positive voltage of 3V or greater is a logic 0 at RX or asserted at a control input. A negative voltage of 3V or greater (more negative) is a logic 1 at RX or Off at a control input. The logic level of an input between -3V and +3V is undefined.

The noise margin, or voltage margin, is the difference between the output and input voltages. RS-232's large voltage swings result in a much wider noise margin than 5V TTL or CMOS logic. For example, an RS-232 output of +5V can attenuate or have noise spikes as large as 2V at the receiver and will still be a valid logic 0. Many RS-232 outputs have wider voltage swings that result in even wider noise margins. The maximum allowed voltage swing is ±15V, though receivers must accept voltages as high as ±25V without damage.

Two other terms you might hear in relation to RS-232 are Mark and Space. On the data lines, Space is logic 0 (positive voltage), and Mark is logic 1 (negative voltage). These names have their roots in the physical marks and spaces mechanical recorders made as they logged binary data.

Timing Limits

The TIA-232 standard includes timing specifications that RS-232-compliant chips must meet.

The specified maximum slew rate limits the bit rate of the interface. Slew rate is a measure of how fast the voltage changes when the output switches and describes an output's instantaneous rate of voltage change. The slew rate of an RS-232 driver must be 30 V/µs or less. Limiting the slew rate improves signal quality by eliminating problems due to voltage reflections that occur on long lines that carry signals with fast rise and fall times. Chapter 7 has more on this topic.

At 30 V/µs, an output requires 0.3 µs to transition from +5V to -5V. RS-232's specified maximum bit rate is 20 kbps, which translates to a bit width of 50 µs, or 16 times as long as the switching time at the fastest allowed slew rate.

In reality, because UARTs read inputs near the middle of the bit and most timing references are very accurate, you can often safely use bit widths as short as 5–10 times the switching time. Some interface chips are rated for operation at 120 or 250 kbps while maintaining slew rates that comply with the standard.

Besides having a maximum switching speed, RS-232 drivers must also meet minimum standards to ensure that signals don't linger in the undefined region between logic states when switching. For control signals and other signals at 40 bps and lower, the line must spend no more than 1 ms in the transition region between a valid logic 0 and logic 1. For other data and timing signals, the limit is 4% of a bit width, which works out to 2 µs at 20 kbps or 0.33 µs at 120 kbps. An output that switches between -5V and +5V in 0.33 µs is changing at the specified upper (fastest) limit for the slew rate. The signals' rise and fall times should also be as nearly equal as possible.

Converting Voltages

Many microcontrollers have asynchronous serial ports that use TTL logic levels based on a 5V power supply or CMOS logic levels based on a power supply of 3V, 3.3V, or 5V. Interfacing to an RS-232 port requires converting between these voltages and RS-232 voltages.

Table 4-3 shows logic levels for different chip families and supply voltages. 5V logic refers to the logic levels used by TTL or CMOS logic chips powered by a single +5V power supply with signal voltages referenced to ground. 3V and 3.3V logic refer to the logic levels used by CMOS logic chips powered by a sin-

Table 4-3: The voltage specifications for logic chips vary with the logic family and power-supply voltage.

Parameter	TTL (5V)	74HC (5V)	74HC (3V)	74HC (3.3V)	74HCT (5V)
Logic-low output (max.) (V)	0.4	0.1	0.1	0.1	0.1
Logic-high output (min.) (V)	2.4	4.9	2.9	3.2	3.5
Logic-low input (max.) (V)	0.8	1	0.6	1.0	0.8
Logic-high input (min.) (V)	2.0	3.5	2.1	2.3	2.0

gle +3V or +3.3V supply with signal voltages referenced to ground. Microcontrollers typically use positive logic, where the more positive logic level, or logic high, is logic 1.

With TTL logic, a logic-low output must be no higher than 0.4V, and a logic-low input must be no higher than 0.8V. A logic-high output must be at least 2.4V, while a logic-high input must be at least 2V. Using these specifications, an interface can have 0.4V of noise without causing errors. These logic levels are specified for the original, standard 7400 series of TTL logic and its derivatives, including 74LS, 74F, and 74ALS TTL.

Most CMOS chips are specified for logic levels with wider noise margins compared to TTL logic. A logic-low CMOS output is no higher than 0.1V, and a logic-low input can be as high as 20% of the power supply, or 1V with a 5V supply. A logic-high output is at least 4.9V, and a logic-high input must be at least 70% of the power supply, or 3.5V with a 5V supply. Families that use these logic levels include the 4000 series, 74HC, and 74AC logic.

Some chips have TTL-compatible inputs and CMOS-compatible outputs. These chips can interface directly with either CMOS or TTL logic. Chips that follow this convention include the 74HCT logic family, many microcontrollers, and the 5V-logic interface on RS-232 and RS-485 driver/receiver chips.

Interface Chips

A convenient way to convert between 3V or 5V logic and RS-232 is to use one of the many chips designed for this purpose. Maxim Integrated Products, Inc. was the first to offer RS-232 interface chips that require only a +5V power supply. Since then, other companies have joined the market, including Intersil Corporation, Linear Technology, National Semiconductor, Sipex Corporation, Texas Instruments, and Zywyn Corporation.

The MAX232

The MAX232 (Figure 4-1) includes two drivers that convert TTL or CMOS inputs to RS-232 outputs and two receivers that convert RS-232 inputs to TTL/CMOS-compatible outputs. The drivers and receivers also invert the signals.

The chip contains two charge-pump voltage converters that act as tiny, unregulated power supplies that enable the chip to support loaded RS-232 outputs of ±5V or greater. Four external capacitors store energy for the supplies. The recommended value for the capacitors is 1µF or larger. If using polarized capacitors, take care to get the polarities correct when you put the circuit together. The voltage at pin 6 is negative, so its capacitor's + terminal connects to ground. Because the outputs can be as high as 10V, be sure the capacitors are rated for a working voltage direct current (WVDC) of at least 15V. (Most are.)

Other Interface Chips

Interface chips to suit just about any application's needs are available.

The MAX232A was an early improvement that uses smaller, 0.1µF charge-pump capacitors and can operate at up to 200 kbps. The MAX233 and MAX233A (Figure 4-1) require no external charge-pump capacitors at all.

The MAX3221 has just one driver and one receiver. For applications that use all eight of the signals in a 9-wire interface, chips are available with three drivers and five receivers (for DTEs) and with five drivers and three receivers (for DCEs).

Speed

Some chips have faster, non-RS-232-compliant slew rates to allow operation at up to 1 Mbps. To support a faster bit rate in both directions, the interfaces at both ends must use faster components. An example of a chip that supports faster bit rates is the MAX3225.

Power-saving Features

Many chips include power-saving features. A Shutdown input can place the chip in a reduced-power mode. Some chips have a separate Enable input that enables the receiver on detecting incoming data even if in shutdown mode.

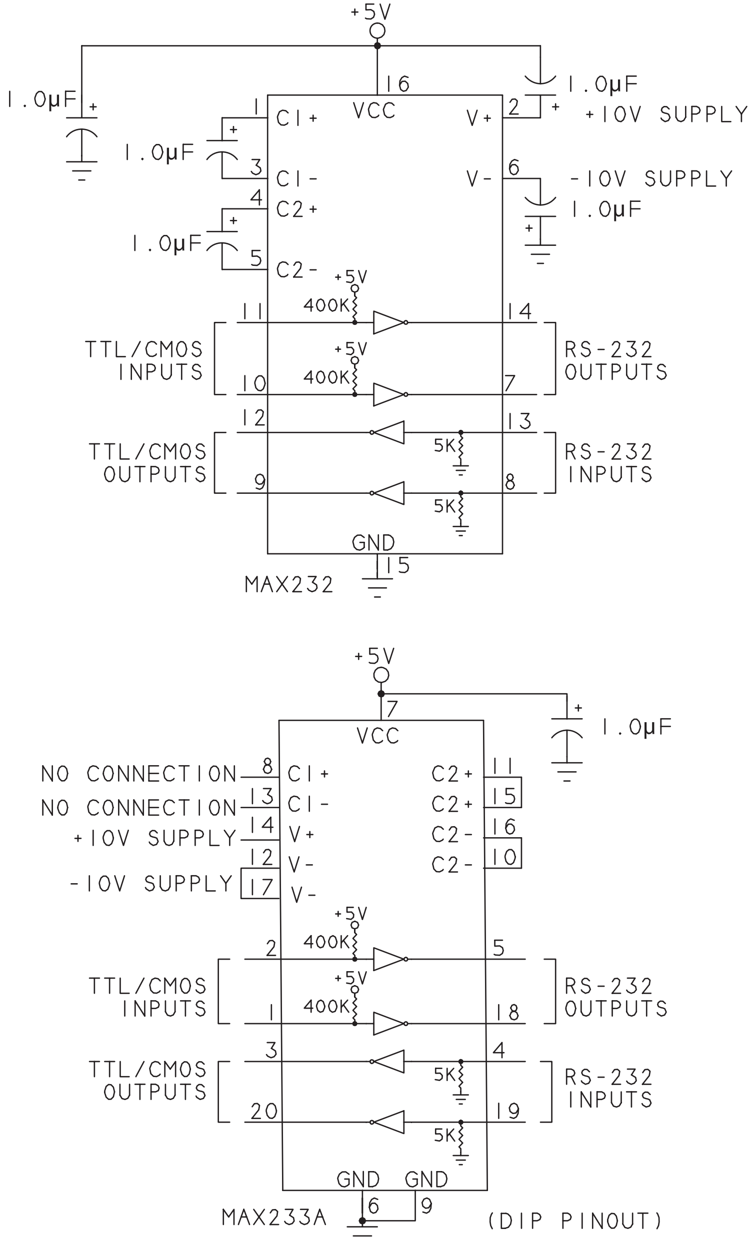


Figure 4-1: Chips like the MAX232 and MAX233A simplify interfacing 5V logic to RS-232.

The MAX3212 has several advanced power-saving features. When all of the RS-232 inputs are less than +3V and greater than -3V, the chip assumes the inputs are open (not connected) and switches to low-power mode. On detecting an input equal to or more positive than +2.7V or equal to or more negative than -2.7V, the chip exits the low-power mode and begins normal operation. Additional inputs enable software to place the chip in and out of low-power mode. The chip's Receiver Enable Control input can save power by placing the outputs of all receivers in a high-impedance state. High-impedance outputs prevent current leaking from the interface chip into a UART or other component whose supply voltage has been removed to save power. When the receivers are high impedance, the chip's Transition Detection output can indicate when a transition has occurred at an input. This output is useful for waking up a microcontroller that is in sleep mode and needs to wake up to process incoming data. The Invalid Signal Detector input can indicate when a a remote port is attached by going high on detecting a valid RS-232 voltage on any receiver.

Other chips can enter low-power mode automatically if there has been no activity for 30 seconds. The MAX3224 is an example.

Voltage Options

An RS-232-compliant chip must have output voltage swings of at least ±5V. To achieve the voltage swings, interface chips can use dual external power supplies, internal charge pumps, inductor circuits, or a combination. For links that don't require RS-232's full ±5V outputs, chips are available with smaller output-voltage swings.

The MAX3386E uses dual charge pumps and a low-dropout transmitter output stage to enables output voltage swings of ±5V or greater even when powered at +3V. The chip also has a Logic-level Supply input that enables interfacing the chip's TTL/CMOS inputs and outputs directly to a UART or another component that has a supply voltage other than +3V.

The MAX3218 operates from a single supply voltage of +1.8V to +4.25V and has output voltage swings of ±5V or greater. The chip requires an external inductor to generate the positive voltage and uses a charge pump to generate the negative voltage. The MAX3316 operates from a single supply voltage of +2.25 to +3V and uses charge pumps to generate output voltage swings of ±3.7V.

Instead of requiring or generating their own negative voltage sources, Maxim/Dallas Semiconductor's DS275 and DS276 borrow the voltage from the interface at the opposite end of the cable.

Circuits that have access to higher-voltage positive and negative supplies don't need voltage converters. The MAX1488E contains four drivers and requires ±12V supplies. The complement to this chip is the MAX1489E, which contains four receivers and operates from a single +5V supply. (The receive circuits don't require higher-voltage supplies.) The MAX3314E requires dual power supplies of ±5V, doesn't use charge pumps, and has RS-232 output swings of slightly less than ±5V.

Protection Options

Many chips incorporate extra protection from electrostatic discharge (ESD) transients on inputs and outputs. Circuits that require electrical isolation can use the MAX250/MAX251 pair with four optocouplers and a transformer.

Short-range Circuits

If you examine the data sheets for the MAX232 and similar chips, you'll see that the specifications for their RS-232 inputs are less stringent than required by the RS-232 standard. As Figure 4-2 shows, the input thresholds are similar to TTL logic, with a logic low input defined as 0.8V or lower and a logic high input defined as 2.0V or higher.

Full Duplex

For cables of 10 ft or less, you may be able to communicate with an RS-232 port by using an inexpensive interface that uses 5V logic rather than RS-232 voltages.

Figure 4-3 shows an option for connecting a 5V port to a remote RS-232 interface. This circuit is intended only for short links because it doesn't meet RS-232's voltage and other specifications.

At the driver, any inverted 5V logic can provide the interface. Figure 4-3 uses Q1, a PN2222 or other NPN general-purpose or switching transistor that functions as an inverter. A TTL/CMOS output drives the base of the transistor, with R1 limiting its base current. When the TTL/CMOS output is low, Q1 is

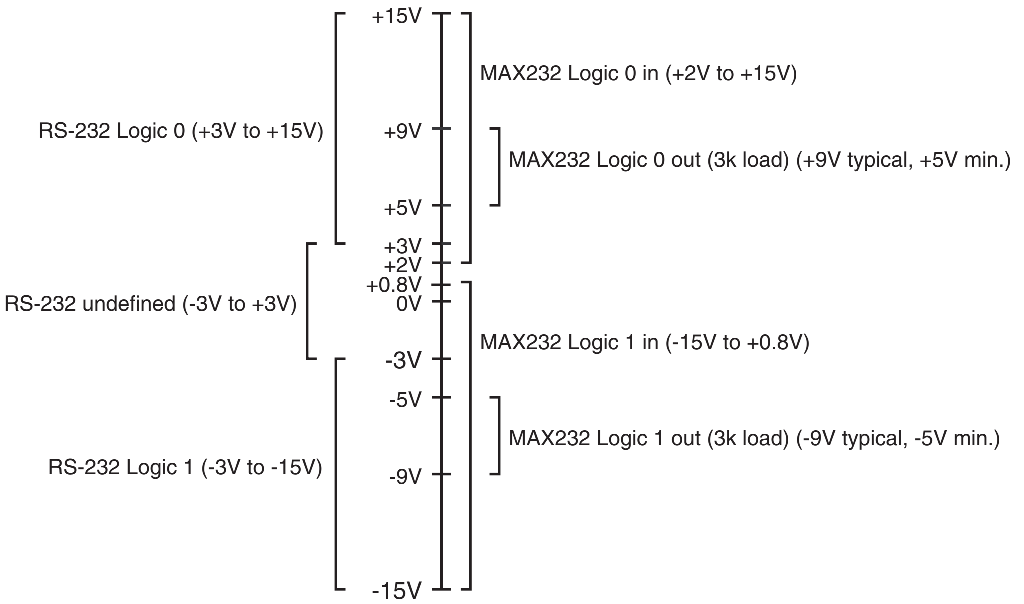


Figure 4-2: The MAX232 and other RS-232 interface chips accept TTL and 5V CMOS logic inputs.

off and R2 pulls RS-232 Out near 5V. When the TTL/CMOS output is high, Q1 switches on, and RS-232 Out is near 0V.

At the receiver, an input designed for use with 5V logic can be damaged by RS-232 voltages, so the circuit must protect the 5V inputs. Transistor Q2 inverts and converts RS-232 voltages to 5V TTL/CMOS levels. The *RS-232 In* signal drives the base of Q2. Resistor R3 limits Q2's base current. Diode D1 protects Q2 by limiting its base voltage base to about -0.7V when *RS-232 In* goes negative. When *RS-232 In* is at or below 0V, Q2 is off and R4 pulls the TTL/CMOS input to 5V. When *RS-232 In* goes positive, Q2 switches on, bringing the TTL/CMOS input low.

Half Duplex

Figure 4-4 shows an alternate 5V circuit that has wider voltage swings than the previous circuit but is useful only in half-duplex links, which transmit in one direction at a time. Parallax, Inc.'s Basic Stamp II uses this type of interface. The negative output matches the negative transmitted voltage, and the positive output is near +5V.

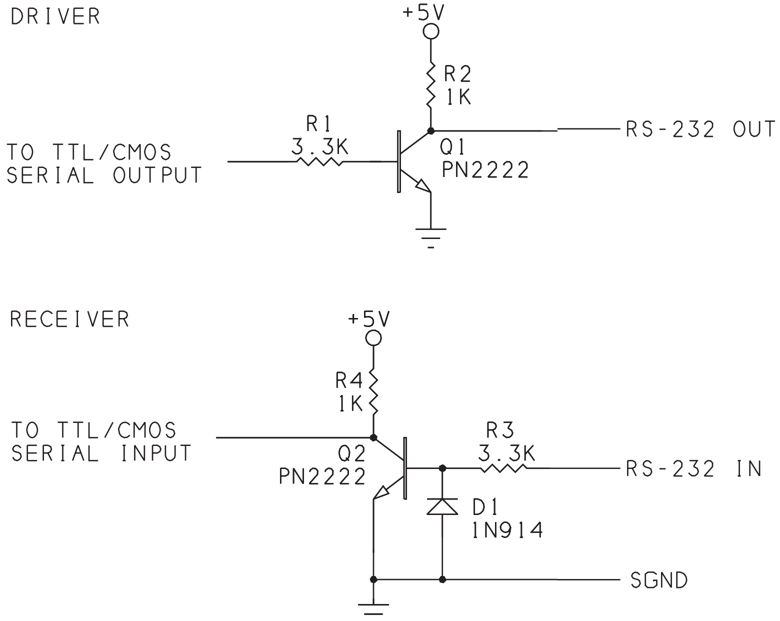


Figure 4-3: This 5V-only interface will work on many short links.

The RS-232 receiver is much like the previous circuit's, but the driver's circuit is different. When *TTL Serial Out* is low, PNP transistor Q1 is on and *RS-232 Out* equals the supply voltage, minus Q1's collector-emitter voltage (a few tenths of a volt). When *TTL Serial Out* is high, Q1 is off, and the RS-232 link loops back on itself through R2. *RS-232 Out* equals *RS-232 In*, minus a small voltage across R2. For this interface to work properly, *RS-232 In* must be idle (negative) whenever *TTL Serial Out* is transmitting.

The NTE2355 and NTE2356 transistors are designed for use as digital switches. They have built-in biasing resistors and a typical maximum frequency of 250 MHz. Their emitter-to-base breakdown voltage is higher than most, at 10V.

Port-powered Circuits

Some low-power circuits that connect to an RS-232 port don't need an external power supply. Instead, they draw all the power they need from the interface itself.

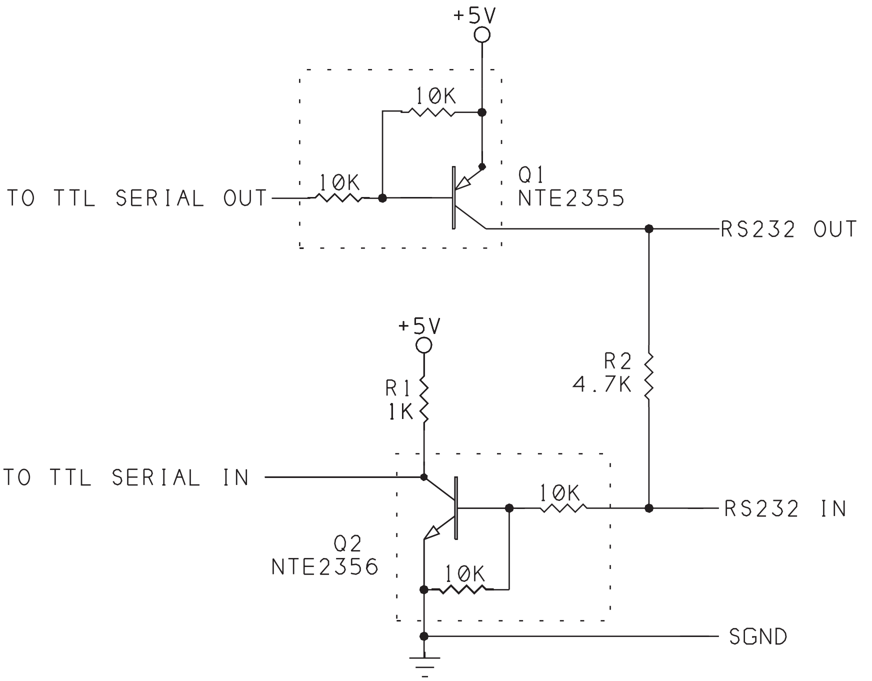


Figure 4-4: This half-duplex interface uses the RS-232 input's negative voltage to pull the RS-232 output below ground.

The power comes from otherwise unused outputs. To comply with the standard, an RS-232 driver's output must be at least 5V with a 3k load. Each output can thus source at least 1.6 mA at 5V. In practice, most RS-232 outputs can source more than this minimum current, but staying within the standard ensures that a circuit will work on any port.

Using Outputs as a Power Source

Figure 4-5 shows ways of using RS-232 outputs as a power source. When asserted, RTS and DTR are between +5 and +15V. Figure 4-5A shows an unregulated output. To double the output current, tie two lines together as shown, with a 1N5819 Schottky diode in each line. The diodes prevent current from feeding back into the interface if the voltages differ. You can use any rectifier diodes, but Schottkys have a lower forward voltage than other silicon diodes. You can even use the TX line as a power source by setting the Break signal, but of course this means you can't use the line for data. However, using TX

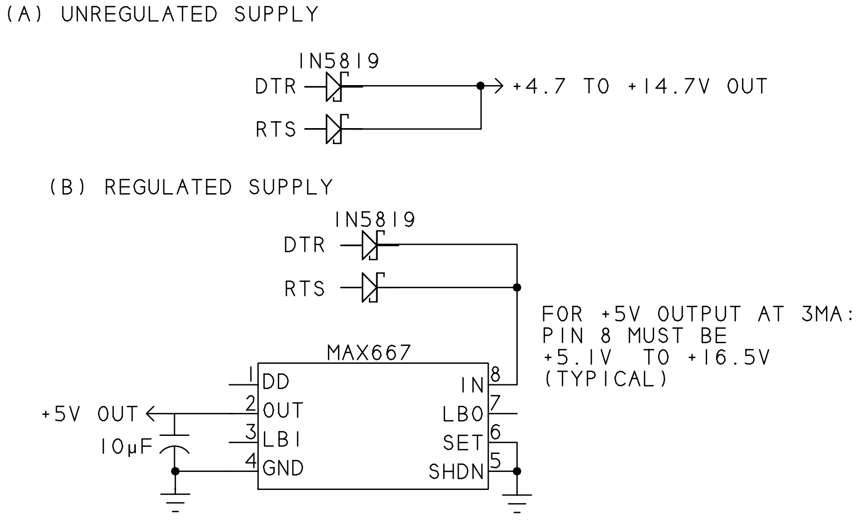


Figure 4-5: You can use spare flow-control outputs as a power source. Diodes enable safely drawing current from multiple outputs. For a regulated voltage, use a high-efficiency regulator like the Max667.

for power might be a solution if you're using the port to control a synchronous interface and using flow-control lines for the clock and data.

Regulating the Voltage

Adding a high-efficiency regulator results in a steady output voltage with little wasted power. Figure 4-5B shows a regulated 5V output using the MAX667 low-dropout, linear regulator. The input can be as high as 16.5V. For an output current of 3 mA, typical operating characteristics specify an input voltage just 10 mV greater than the output, so the circuit will work with most ports. The regulator's quiescent current is under 100 µA with a load of several mA.

For a circuit that will work even if the RS-232 output drops below 5V, use the MAX770, a switching regulator that has a 5V output with an input between 2V and 16V. The MAX770 requires several external components, including an inductor and output transistor.

Another option for port power is to use a lower regulated voltage, either by connecting voltage-divider resistors between Vout, Set, and Gnd as described in the MAX667's data sheet or by using a regulator with a lower fixed output such as the 3V MAX689.

Because you can count on getting at most a few milliamperes from the port, use the lowest-power components you can find and a 3V supply if possible.

Alternate Interfaces

If RS-232 doesn't meet your circuit's needs, an alternate interface might do the job. For some applications, a direct connection or 5V buffers and drivers are all that's needed. Or a different standard interface might be more appropriate.

Direct Connection

If the interface is between two microcontrollers or other chips whose serial ports use 3V or 5V logic, you may be able to connect the ports directly, output to input, without using RS-232 at all. The outputs on some microcontrollers, such as the 8051, are too weak to drive a cable any useful distance. In that case, you can use a buffer/line driver such as the 74HCT541. These and similar chips can drive cables up to 10–15 ft in many environments. At the receivers, Schmitt-trigger inputs help to reject noise. These TTL/CMOS buffer/drivers are cheaper than RS-232 interface chips.

Other Unbalanced Interfaces

Table 4-4 compares RS-232 with other TIA interfaces that use unbalanced lines. Chapter 6 has details on TIA's balanced interfaces.

TIA-562 defines an interface for transmitting at up 64 kbps. The receiver sensitivity is identical to RS-232, but the output voltages are slightly lower, with a range from ±3.3V to ±13.2V. For data rates faster than 20 kbps, maximum capacitance must be 1000 pF or less. Linear Technology's LTC1385 is a TIA-562 interface chip that operates from a single 3.3V supply.

Other alternatives use a combination of balanced and unbalanced signals. As Figure 4-6 shows, TIA/EIA-423 (commonly called RS-423) allows up to ten receivers and one transmitter. The drivers are unbalanced, like RS-232, but the receivers are balanced and are identical to RS-422's receivers, described in Chapter 6. The receivers obtain their differential inputs using the driver's output voltage and the cable's signal-ground wire.

TIA-530 uses balanced drivers and receivers for TX, RX, RTS, CTS, and CD, and unbalanced lines for DTR, DSR, and RI. The result is better performance than RS-232 at a cost of more wires in the cable. Another source for similar standards is ITU/CCITT.

Table 4-4: In addition to RS-232, there are several other EIA/TIA interfaces for unbalanced lines.

Specification	TIA-232-F	TIA/EIA-423-B	TIA-562	TIA-530-A
Cable length, max (ft), unshielded cable, 20pF/ft, 100kbps	50	50	15 ft @ 64kbps	4000
Data rate, max (bps)	20k	100k	64k	2.1M
Driver output (minimum, V)	± 5	± 3.6	±3.3	± 3.3, ±2*
Driver output (maximum, V)	± 15	± 6	±13.2	±6, ±10*
Receiver sensitivity (V)	± 3	± 0.2	±3	±0.2
Maximum number of drivers	1	1	1	1
Maximum number receivers	1	10	1	10
Receiver input resistance (Ω)	3k-7k	450 (minimum)	3k-7k	450, 4k*
*Data and some control lines use a balanced interface. Other signals use an unbalanced interface.				

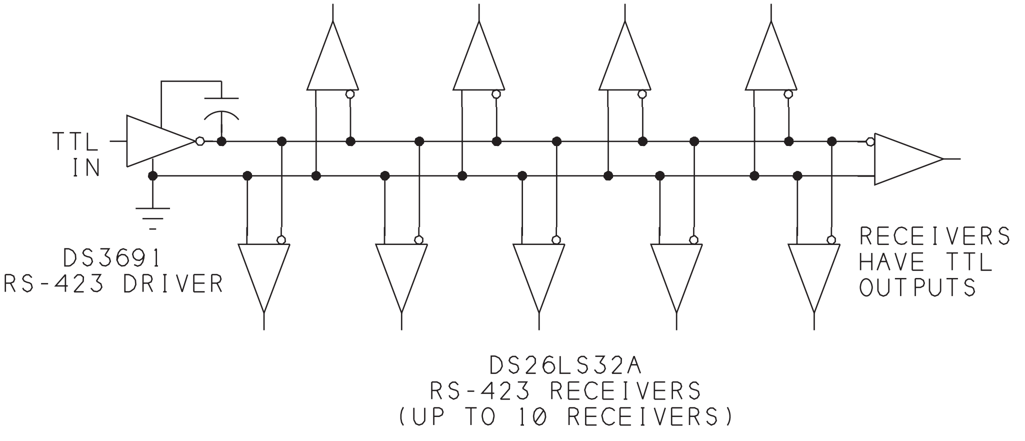


Figure 4-6: An RS-423 interface can have just one transmitter but can have up to ten receivers.

This page intentionally left blank

5

Designing RS-232 Links

This chapter discusses RS-232 cables, connectors, wiring configurations, and options for isolating an interface.

Connectors and Adapters

An RS-232 interface can be configured as a DTE or DCE. In addition, RS-232 interfaces may use any of a variety of connector types. Adapter modules enable two interfaces to connect to each other no matter what combination of DTE and DCE and connector types the interfaces start out with.

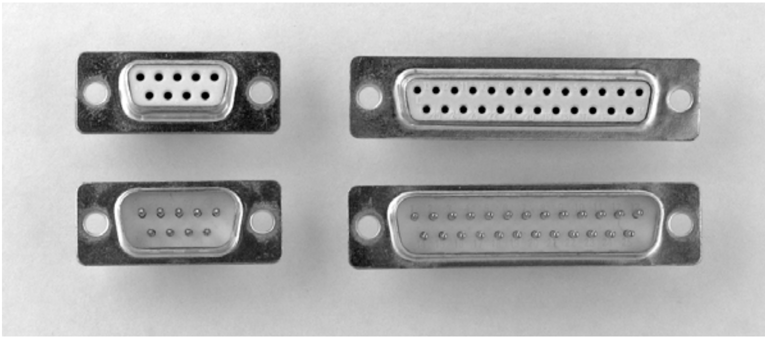


Figure 5-1: RS-232 D-sub connectors: clockwise from top left: 9-pin female, 25-pin female, 25-pin male, 9-pin male.

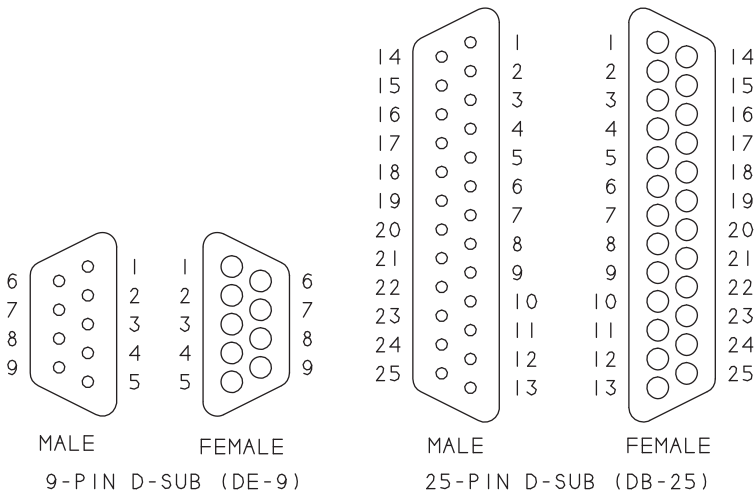


Figure 5-2: Pin and socket locations for RS-232 D-sub connectors as seen from the mating side of the connector.

Connector Options

Figure 5-1 shows the 9- and 25-pin D-sub (subminiature “D”) connectors that many RS-232 interfaces use. Figure 5-2 shows the pinouts of the connectors. On most connectors, the pin or socket numbers are stamped near the pins or sockets, though you may need to look closely to see the numbers.

TIA-232 specifies using the 25-pin D-sub connector or a compact 26-pin Alt A connectors. These connector types are rarely used in recent designs because vir-



Figure 5-3: An RJ-11 (left) or RJ-45 (right) modular connector can be an option for interfaces of up to eight lines.

tually no applications need more than 9 wires. The TIA-574 standard documents the pinout for the 9-pin connector.

Other names sometimes used for the 9-pin D-sub connector are DE-9, with the E indicating the shell size, and DB-9, which is technically incorrect because it suggests the connector uses the 25-pin shell size.

Some RS-232 interfaces use the same modular plugs and jacks used for indoor telephone wiring or Ethernet networking (Figure 5-3). These connectors are compact, reliable, and inexpensive for applications that don't need nine wires. The phone connector commonly called RJ-11 has 6 contacts. Standard phone cables typically have 2 or 4 wires with the extra pins left open. The Ethernet connector commonly called RJ-45 has eight contacts. The TIA-561 standard specifies a pinout for an 8-contact connector that includes everything in the 9-pin interface except DSR (Table 5-1). The 6-position connector has no standard pinout for serial communications.

Adapters

If you need to attach different connector types to each other, adapter modules and cables are available in a variety of configurations, or you can make your own. Sources for adapter modules include B&B Electronics Manufacturing Company (*www.bb-elec.com*) and Jameco Electronics *(www.jameco.com*).

Table 5-1: TIA-561 provides this recommended pinout for use with an 8-position RJ-type connector.

Pin	Signal
1	RI
2	CD
3	DTR
4	SG
5	RX
6	TX
7	CTS
8	RTS

9-pins to 25-pin Links

To connect a DTE to a DCE with a 9-pin D-sub at each end, the cable connects each wire straight across, pin 1 to pin 1, pin 2 to pin 2, and so on. Connecting a 9-pin D-sub to a 25-pin D-sub requires a cable or adapter that routes the signals correctly (Figure 5-4).

Gender Changers

A gender changer is a adapter that enables two male (pin) connectors or two female (receptacle) connectors to connect to each other. In an ideal world, every DTE would use a male connector, and every DCE would use a female connector. If you happen to have a situation where you need to connect a DTE and DCE whose connectors have the same gender, an adapter called a gender

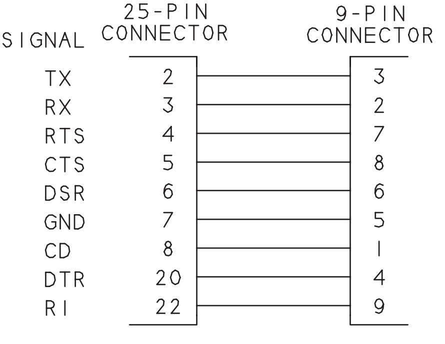


Figure 5-4: Use this wiring to convert between 9-pin and 25-pin connectors.

changer can help. To connect two male connectors, use a female-to-female gender changer, which has receptacles on both ends. To connect two female connectors, use a male-to-male gender changer, which has pins on both ends.

Gender changers connect the RS-232 lines straight across: TX to TX, RX to RX, and so on. A gender changer can have the same number of pins on each end, or the adapter can convert between 9- and 25-pin D-subs as in Figure 5-4.

Null Modems

Some lines connect two ports configured as DTEs or, less often, two ports configured as DCEs. If you use a straight-across cable, the two RX inputs connect to each other and neither port receives anything sent by the other port.

The solution is to use a null-modem adapter or a cable that provides the null-modem connections. The adapter simulates a connection between a DTE and a DCE by routing each output to its corresponding input. For example, each TX output connects to RX at the other end of the cable or adapter. The name null modem refers to its origin as a cable that bypasses the computer-to-modem (DTE-to-DCE) connection and directly connects two computers (DTE-to-DTE).

Figure 5-5 shows several null-modem configurations. Interfaces that don't use flow-control lines can use a 3-wire null modem that only swaps the TX and RX lines. A null modem with flow control also connects RTS to CTS and DTR to DSR. On an adapter for use with two ports configured as DTEs, the CD and DSR inputs are often tied together to cause the CD input to follow the remote computer's DTR output. A third type of null modem uses loop-back flow control, connecting each flow-control output to a corresponding input on the same end. On a port configured as a DTE, the DTR output can also drive the CD input. The loop-back adapter emulates a remote device that is always ready to receive data. This type of null modem can be useful when a computer using flow control communicates with a device that doesn't support flow control. Two ports configured as DCEs can connect to each other using any of the null-modem configurations except that CD should be left open on both ports.

Using Microcontroller Development Boards

Many microcontroller manufacturers and other vendors offer development boards with the components for a generic system, often including an RS-232 converter and connector.

A typical microcontroller has a serial output labeled TX and a serial input labeled RX. The RS-232 port on development boards is typically configured as a DCE, where TX is an input and RX is an output. The microcontroller's TX output thus controls the RS-232 RX output, and the RS-232 TX output controls the microcontroller's RX input.

Using flow-control signals on a development board sometimes requires cutting hard-wired connections and adding wires to connect flow-control signals to the RS-232 connector.

An example of a development board is Microchip's PICDEM™ 2 Plus. The board has a MAX232 converter with a driver/receiver pair that interfaces the

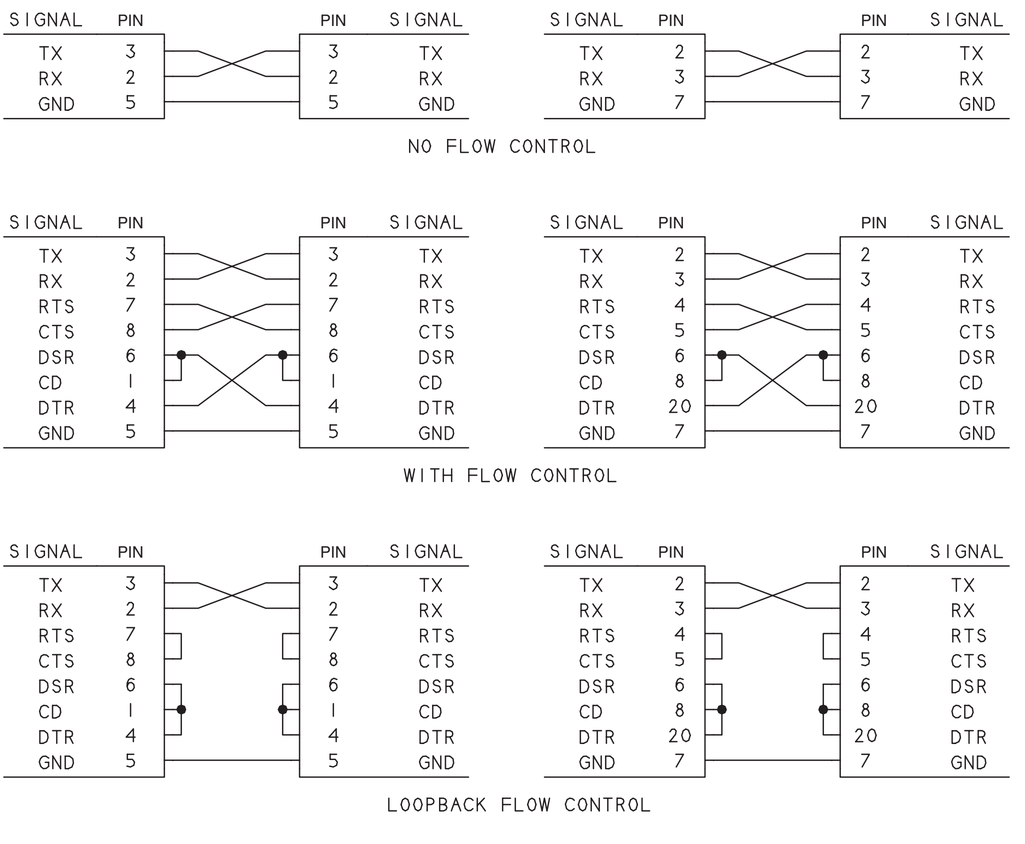


Figure 5-5: Two ports configured as DTEs can connect to each other using any of these null-modem adapters. On the left are 9-pin connections; on the right are 25-pin connections.

microcontroller's TX and RX pins to RX and TX on the RS-232 connector. The MAX232's other driver/receiver pair is unused.

To add flow-control lines, you can use the MAX232's spare driver and receiver to interface port bits on the microcontroller to the RS-232 connector. The on-board MAX232 is in a surface-mount package, however, so soldering jumper wires is difficult. Another option is to add another MAX232 or similar interface chip in a through-hole package to the board's prototyping area and wire the needed connections to the chip.

The PICDEM 2 Plus board connects DTR to DSR on the RS-232 connector. To enable using DTR and DSR for flow control (or other uses), cut the circuit-board trace that connects these signals together and add jumper wires to connect an RS-232 driver and receiver to microcontroller port pins and the connector.

Other development boards connect CTS and RTS together at the RS-232 connector. If you need to use hardware flow control, check your board's schematic and modify the circuits as needed.

Cables

RS-232 cables can vary in length, number of wires, and shielding.

Length Limits

Early versions of the RS-232 standard recommended limiting cable length to 50 ft, and this is still a good general guideline. For data rates of 20 kbps or less, you can use just about any type of cable of this length or less.

Later versions of the standard instead specify a maximum capacitance of 2500 pF at the receiver. This value includes the capacitance of the receiver, the mutual capacitance between conductors in the cable, and the capacitance between the conductor and the cable shield or, on unshielded cable, between the conductor and earth ground.

The capacitance has several effects. It limits the slew rate, with a larger capacitance resulting in a lower slew rate and slower transitions. A higher capacitance means that a voltage change requires more current to charge the capacitance, so the overall power consumption of the drivers is greater. Capacitance between wires can result in crosstalk, where the signal on one wire also shows up on adjacent wires.

Unshielded Cable

Cable manufacturers often specify the capacitance of their products in pF/ft. For unshielded cable, an appendix to TIA-232 recommends adding 50% to the cable's capacitance to account for conductor-to-ground capacitance.

The formula to calculate cable length for unshielded cable is:

```
CableLength = (2500 - InputCapacitanceOfReceiver) / (CableCapacitance * 1.5)
```

CableLength is in ft, InputCapacitanceOfReceiver is in pF, and CableCapacitance is in pF/ft.

TIA-232 doesn't recommend a cable type. Typical capacitance of ribbon cable is 15 pF/ft. Assuming that the receiver's input is 100 pF, the cable can be as long as 106 feet ((2500-100)/(15*1.5)). Typical capacitance for a single, unshielded twisted pair is 12pF/ft. Again assuming an input capacitance of 100 pF, maximum cable length is 133 feet.

Twisted Pairs

For reduced crosstalk, you can use twisted-pair cable with each pair containing a signal wire and a ground wire. Chapter 7 has more on twisted pairs.

Shielded Cable

Adding shielding to the cable shortens the maximum allowed length, but shielding can block noise from coupling into or out of the cable. For shielded twisted-pair cable, an appendix to TIA-232 recommends tripling the value of the conductor-to-conductor capacitance to account for the conductor-to-shield capacitance.

This is the formula to calculate cable length for shielded cable:

```
CableLength = (2500 - InputCapacitanceOfReceiver) / (CableCapacitance * 3)
```

CableLength is in ft, InputCapacitanceOfReceiver is in pF, and CableCapacitance is in pF/ft.

The maximum length of shielded, twisted-pair cable is thus around 66 feet.

If you want to use a cable that exceeds the capacitance limit, you'll probably still be able to communicate, though at lower bit rates. Over short cables with correspondingly lower capacitance, you should be able to communicate faster than 20 kbps if the transmitting and receiving hardware supports higher rates.

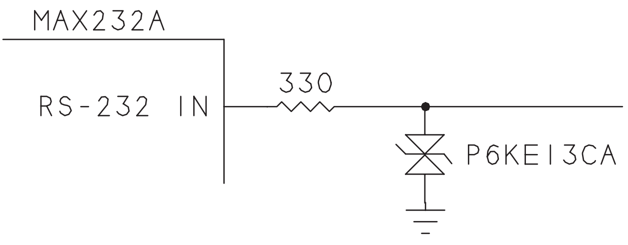


Figure 5-6: A TVS diode can protect components from high voltages on the RS-232 line.

Surge Protection

A way to protect circuits from noise or damaging voltages and currents is to use surge protection. The ideal surge protection absorbs all voltages and currents outside the circuits' operating range while not limiting transmissions of valid signals in any way. In real life, a variety of devices can protect a link from many disasters due to voltage surges, though all add some capacitance to the link and thus limit the maximum bit rate and cable length.

In normal operation, a protection device presents a high impedance and is virtually invisible to the transmitting circuits. When the line sees a high-voltage surge, the protection device switches on and provides a low-impedance path to ground.

Two surge-suppression devices used in RS-232 lines are transient voltage suppression (TVS) diodes and gas-discharge tubes. TVS diodes are bidirectional zener diodes that have low capacitance when off, respond quickly (1 ps), and are available in many breakdown-voltage ranges. Gas-discharge tubes are slower but can protect against higher voltages. Some circuits use both.

One terminal of the suppression component connects to a data line as close as possible to the circuits being protected. The other terminal connects to a ground strap or other low-impedance connection directly to an earth ground (Figure 5-6). The component's breakdown voltage should be 10–15% greater higher than the normal voltages on the line. A series resistor at the RS-232 input can limit the input current.

Isolated Lines

RS-232's generous noise margins help make the interface reliable and immune to data errors caused by external noise coupling into the wires. When a line needs more protection, isolation can keep noise from coupling between a cable's wires and the circuits they connect to.

Isolation works by dividing a circuit into sections that have no galvanic connection. Optical and magnetic coupling can transfer data and power between the sections and filter out noise.

Isolation can isolate the grounds, the data lines, or both. Ground isolation makes a circuit immune to power surges and noise in the earth ground shared by nearby circuits. With long cables, ground isolation also makes the line immune to differences in ground potential from end to end. Isolating the data lines keeps noise from coupling between the lines and the circuits they connect to.

Ways to Achieve Isolation

Most circuit connections use solder joints or mechanical connections such screw terminals or crimps. These connections create an ohmic path that enables current to flow between components. Galvanically isolated circuits can use optical or magnetic coupling to transfer power and data.

Common ways to achieve galvanic isolation include transformers to isolate power and optoisolators to isolate data. In a transformer, magnetic coupling between the windings causes current in the primary winding to induce a current in the secondary winding. Optoisolators transfer energy via phototransistors and photodiodes that emit and detect energy in the visible or infrared spectrum. In a similar way, a fiber-optic interface converts an electrical signal to light for transmitting in an optical fiber and converts light to an electrical signal at the receiver.

For complete isolation, each end of an RS-232 line requires an isolated power supply for the RS-232 interface and an interface to transfer the data across the isolation barrier.

About Grounds

Understanding isolation requires understanding the concept of ground. All current must eventually return to its source. A ground connection is any low-impedance path that serves this purpose. Different types of grounds

include signal ground, analog and digital grounds, earth ground, and safety ground.

Signal Ground

Signal ground (RS-232's SG pin) refers to the ground terminal of a power supply's output, and all points that connect to it. Because RS-232 receivers measure voltages between the signal lines and SG, a noise spike on the SG line can cause a receiver to misread a logic level.

When a circuit uses more than one power supply, even if the supplies' grounds aren't isolated from each other, maintaining separate ground paths reduces the noise that couples from one path into another. The ground wires of each supply can use separate wiring and circuit-board traces and connect together only at the supplies.

Circuits that contain both analog and digital circuits can provide a separate ground path for each, connecting the two paths at only one point near the power supply. Digital grounds tend to be noisy because digital outputs draw high currents when they switch. Analog circuits can be sensitive to small voltage changes and often use a separate ground path from digital circuits in the same device.

Safety Ground

Safety ground, or protective ground, is a connection to earth ground and is commonly a large-diameter copper wire or copper-plated pipe partially buried underground. One of the three wires at an electrical outlet's wall socket connects to a safety ground.

The other wires at the outlet are the hot wire, which carries the line voltage (115VAC in the U.S.), and the neutral wire, which carries the return current. The neutral wire connects to the safety ground at the service entrance to the building. The neutral wires of all of a building's circuits normally have a common connection at the service entrance.

The safety ground provides a low-impedance path to ground in case of a fault. For example, in many power supplies, a screw terminal connects the safety-ground wire to the supply's metal chassis. If the chassis isn't grounded and a loose wire or component failure causes a voltage source to contact the chassis, the chassis may carry a high voltage. Someone who touches the chassis while in contact with electrical ground will receive an electrical shock. If the

chassis is grounded, current instead follows the low-impedance path to earth ground until a fuse blows and the circuit opens, removing the danger.

TIA-232 says that a DCE can have a removable strap to connect SG to safety ground. In practice, the SG line on both DCEs and DTEs often connects to a safety ground.

Earth Ground

Earth ground refers to the electrical potential of the earth itself. Because any electrical circuit may connect to earth ground, it's usually not a quiet, stable reference and can carry huge amounts of noise of all types. Events that can cause ground noise include equipment switching on and off, power-system fluctuations, circuit malfunctions, lightning, or anything else that causes a surge in current. The noise can show up as dips, spikes, 60-Hz oscillations, or just about any other variation.

Earth grounds at different locations don't always connect electrically to each other. Whether the grounds connect and how much the ground voltages vary depends partly on how well the medium between the ground connections conducts electricity. Within a building, the electrical wiring provides a common connection to earth ground. Between buildings or over long distances, current will follow whatever path it can find. Wet soil is a better conductor than solid granite.

Effects of Common Grounds

If both ends of an RS-232 cable share a common earth ground and the SG wire also connects to safety ground, ground currents from all sources will choose the path of least resistance: earth ground or the SG wire. The situation where there are multiple return paths is called a ground loop and is not desirable. If the two devices are in different buildings and using different power systems, SG is likely to have lower impedance than other paths, and ground currents from other sources may find their way into the cable's ground wire. The result is a noisy ground. A line with isolated grounds avoids this problem.

Power Supply Grounds

An isolated interface requires a power supply for each side of the isolation barrier. Figure 5-7 shows two isolated RS-232 interfaces. Each uses a dual power supply where a transformer steps 115VAC on the primary winding to lower

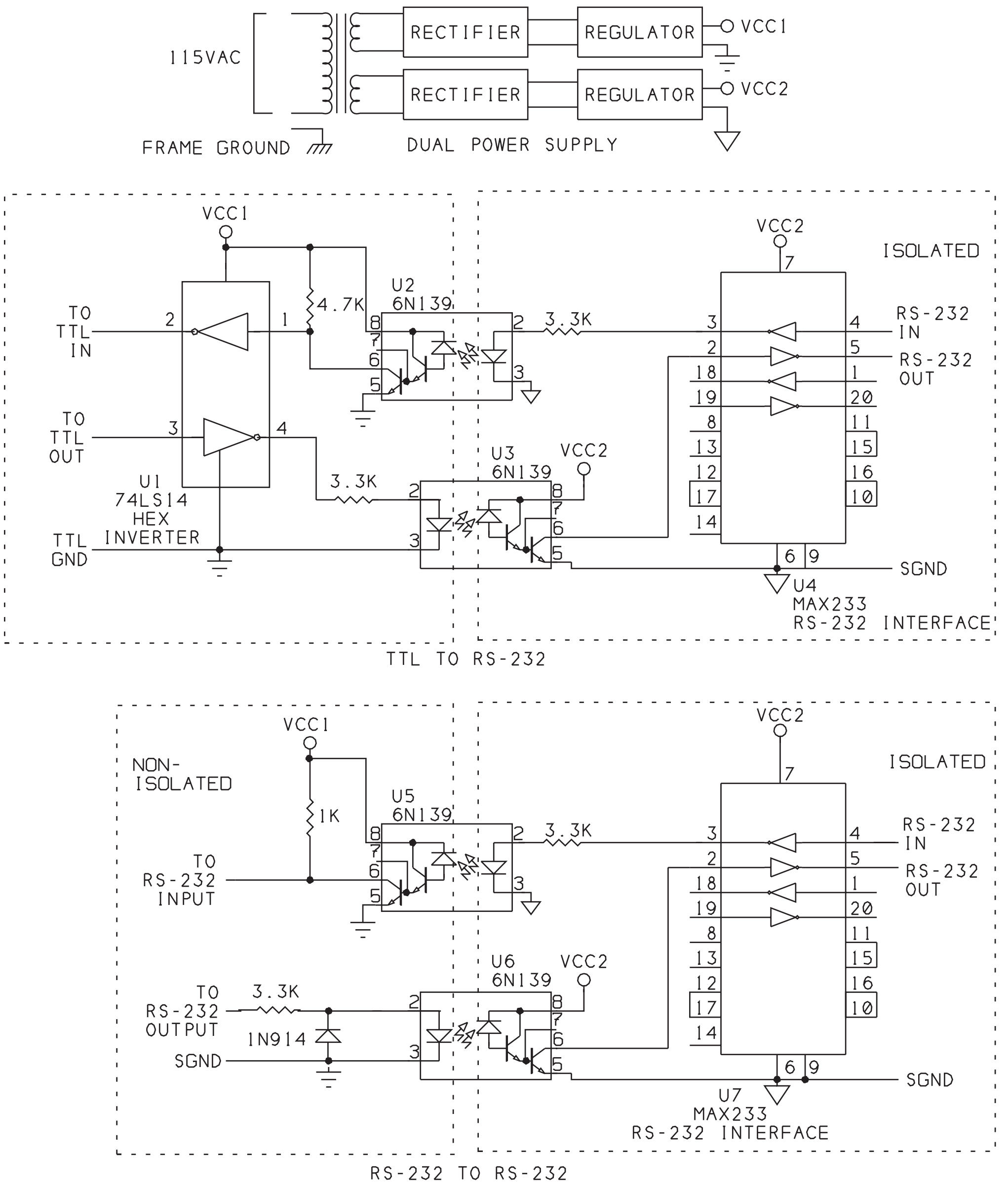


Figure 5-7: Optoisolators can isolate a TTL or RS-232 interface.

voltages on two secondary windings. One winding provides voltage for the computer or other circuits that connect to equipment side of the optoisolator. The other winding provides the voltage for the RS-232 interface. Each supply has its own ground, and the grounds must have no common connection to an earth ground, the chassis, or signal ground. Instead of one supply with two windings, an interface can use two entirely separate power supplies or batteries whose outputs don't share a common ground.

To determine if a DC supply is isolated from earth ground, you need to know something about what's inside the supply.

In most DC supplies powered by line voltage, a transformer steps the line voltage to a lower value, and other components rectify, filter, and regulate the transformer's output to a steady DC value. The only connection required between the transformer's primary and secondary windings is the magnetic coupling induced when current flows in the primary. The transformer thus has the ability to isolate the power supply's outputs from the line-voltage wiring and safety ground.

In fact, the outputs of some power supplies for digital circuits have no connection to safety ground. There is little risk of electrical shock at the outputs because the voltages are low, the regulator limits the current, and a fuse opens the circuit if it tries to draw large currents.

In other supplies, the output's ground terminal connects to safety ground, breaking the isolation. The result is a shared ground with any other circuits that also connect to the safety ground, or earth ground. A connection can exist even if the circuits are in different buildings or thousands of feet apart.

The safest route is to assume that a supply's ground isn't isolated unless you can prove that it is isolated. Don't assume that the SG pin on a PC's RS-232 or RS-485 port is isolated from earth ground.

A supply with a 2-wire power plug may appear to have no safety-ground connection, but the neutral wire connects to safety ground when the supply is plugged in. The supply's output is isolated only if its ground line doesn't connect to the neutral wire.

For supplies that contain a transformer, you can use an ohmmeter to find out if the output is isolated from safety ground. *With the supply unplugged from the wall socket or other power source,* measure the resistance between safety ground on the supply's AC power plug and the DC output's ground terminal. If the meter shows a measurable resistance, the output isn't isolated. The neutral wire

and safety ground should have no connection inside the supply. You can verify this lack of connection with an ohmmeter as well.

Some supplies don't use transformers. The supply directly rectifies, reduces, and filters the line voltage. In this case, the output isn't isolated from earth ground. Even if the power plug has no safety-ground pin, the neutral wire connects to safety ground when the supply is plugged in.

Optoisolating

Optoisolators transfer signals across an isolation barrier. An optoisolator consists of a photodiode coupled to a phototransistor. Current through the photodiode causes the photodiode to emit energy in the visible or infrared spectrum. The energy switches the phototransistor on, creating a low resistance between the transistor's emitter and collector. The phototransistor's base can be left unconnected. Adding a resistor from base to emitter results in faster switching but lower output current.

The interfaces in Figure 5-7 use 6N139 optoisolators, which are designed for direct interfacing to LSTTL logic. Their gain is high: 400% with a photodiode current of just 0.5 mA. In the TTL-to-RS-232 circuit, a logic low at pin 3 of the 74LS14 inverter causes current to flow through the photodiode. The current switches on the corresponding phototransistor, bringing its collector low. The MAX233 inverts the signal and transmits a positive RS-232 voltage.

A logic high on pin 3 of the 74LS14 switches the photodiode and phototransistor off. The MAX233's internal pull up at pin 2 results in a negative RS-232 voltage.

The other direction works in a similar way. A negative RS-232 input causes the MAX233 to output a logic high. This signal switches the photodiode and its phototransistor on, resulting in a logic low at pin 1 and a logic high at pin 2 of the 74LS14. A positive RS-232 input causes the MAX233 to output a logic low. The logic low switches the photodiode and its phototransistor off. A pull up brings pin 1 of the 74LS14 high, resulting in a logic low at pin 2.

The RS-232-to-RS-232 circuit shows how to isolate an existing, non-isolated RS-232 interface by using an RS-232 output to drive a photodiode directly. When the RS-232 voltage is positive, the photodiode is on, and the isolated RS-232 output is also positive. When the non-isolated output is negative, the photodiode is off, a 1N914 diode clamps the voltage at about -0.7V, and the isolated RS-232 output is negative. In the other direction, the circuits are simi-

lar to the TTL-to-RS-232 circuit except that no 74LS14 inverter is needed because the non-isolated RS-232 interface inverts the signal.

For VCC1 in the RS-232 to RS-232 circuit, you can use a positive output at DTR or RTS if the signal is otherwise unused. The cable on the VCC1 side of this circuit should be short.

Typical turn-on and turn-off times for phototransistors is several microseconds, which should cause no problems at data rates of 20 kbps or less. For fast bit rates, look for a photodiode with switching times of 1/10 or less of the bit width.

Another way to achieve an isolated interface is to use separate, isolated supplies for the RS-232 voltages.

Maxim Integrated Products' MAX250 and MAX251 enable creating an isolated interface with the addition of four optocouplers and a transformer.

Fiber Optics

A different way to isolate a link is to use fiber-optic cable in place of copper wire. Fiber-optic cable encodes data as the presence or absence of light or via more complex encoding methods.

Fiber-optic cable has several advantages. The signals are immune to ground noise and electromagnetic interference and generate no electromagnetic interface. A cable typically can run 1 to 2.5 miles before requiring a repeater. The main disadvantage is expense, including the need for special connectors and tools.

Debugging Tools

A breakout box, voltmeter, and oscilloscope can be helpful when setting up and debugging circuits for serial communications.

Using a Breakout Box

A breakout box (Figure 5-8) connects in series with an RS-232 cable and displays the status of each line in the cable. LEDs indicate the logic states of the signals. Some boxes also have switches and jumper-wire connections that enable you to rewire the interface in any configuration. For example, you can experi-

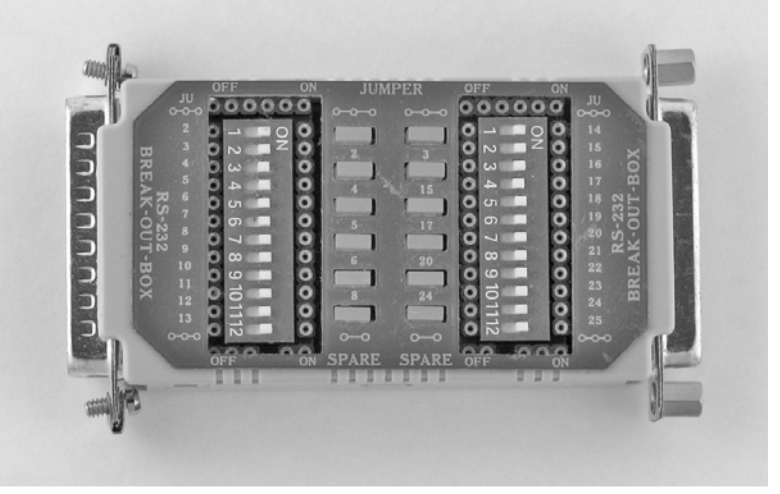


Figure 5-8: Breakout boxes often include LEDs that show the signal states and jumpers or switches to enable different wiring configurations. This breakout box is from B&B Electronics Manufacturing Company (www.bb-elec.com).

ment with different connections to determine what type of null modem a link requires.

Monitoring with a Voltmeter

If you lack a breakout box, you can monitor lines with a voltmeter. If you're not sure whether a connector is wired as a DTE or DCE, a voltmeter is all you need to identify which of pins 2 and 3 is the data input, and which is the data output.

Measure on a port that is powered, but idle (not currently transmitting or receiving). On the connector, measure the voltage from pin 2 to signal ground (pin 5 on a 9-pin connector). Also measure from pin 3 to signal ground. On an idle port, an output should be a negative voltage of at least -5V. An open, or unconnected, input is typically near 0V and in any case should measure between -3V and +3V.

For example, on a DB-9 connector, if pin 2 has a negative voltage, the interface is a DCE, and if pin 3 has a negative voltage, the interface is a DTE.

Oscilloscopes and Logic Analyzers

Sometimes there's no substitute for viewing the waveforms. A digital oscilloscope is handy for viewing serial data. You can trigger on a Start bit or control signal and the scope will display and preserve the waveform to examine at your leisure. You can save a waveform and compare it to waveforms captured later.

When viewing RS-232 data, don't forget that the data transmits least significant bit first and that the logic levels are inverted. If the bits following the Start bit are 11111110 from left to right on the screen, the value of the byte is 80h, not FEh or 7Fh.

A logic analyzer is another tool for viewing serial data. Many logic analyzers have eight or more channels so you can view multiple data and flow-control lines at once. Also available are hardware and software data analyzers and testers designed for debugging serial communications. These tools include capabilities such as decoding data using various formats, triggering on specific characters, and displaying statistics. Microsoft provides a free Portmon utility that displays serial-port activity at the driver level. (Search *microsoft.com* for "portmon".)

If you're using a USB/serial converter and happen to have a USB protocol analyzer, the analyzer provides another way to view COM-port data.

6

Inside RS-485

RS-485 is a solution for applications that need to communicate over longer distances or at higher speeds than RS-232 can handle. RS-485 also isn't limited to two devices. An RS-485 network can connect as many as 256 computers along a single pair of wires. This chapter introduces RS-485 signals and interfacing.

About RS-485

The interface commonly known as RS-485 is defined by *TIA-485-A: Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems*. A similar standard is ISO/IEC 8482.1993. A supplementary document is *TSB-89-A: Application Guidelines for TIA/EIA-485-A*.

RS-485 has several advantages over RS-232:

- Low cost. The drivers and receivers are inexpensive and require just a single +5V (or lower) supply to generate the required minimum 1.5V difference at the differential outputs. In contrast, RS-232's minimum output swing of ±5V requires dual supplies or expensive interface chips to generate the voltages.

- Networking ability. Instead of being limited to two devices, RS-485 is a multidrop interface that can have multiple drivers and receivers. With high-impedance receivers, an RS-485 network can have up to 256 nodes.
- Long links. An RS-485 cable can be as long as 4000 ft compared to RS-232's typical limit of 50 to 130 ft.
- Speed. The bit rate can be as high as 10 Mbps.

The cable length and bit rate are related. Lower bit rates allow longer cables.

Table 6-1 shows specifications for RS-485 and a related interface, RS-422, which is limited to one driver and ten receivers but allows greater differential voltages at inputs.

Unlike RS-232, RS-485 has no defined hardware lines for flow control. Some applications can use Xon/Xoff flow-control codes. When not using flow control, software should use the alternate methods described in Chapter 2 to ensure that the receive buffer doesn't overflow.

Balanced and Unbalanced Lines

The main reason why RS-485 can transmit over long distances is its use of balanced lines, which have excellent noise immunity. Each signal has a dedicated pair of wires. The voltage on one wire equals the negative, or complement, of the voltage on the other wire. The receiver detects the difference between the voltages. Figure 6-1 illustrates. Another term for this type of transmission is differential signaling.

TIA-485-A designates the two lines in a differential pair as A and B. At a typical RS-485 driver, a TTL logic-high input brings line A more positive than line B, while a TTL logic-low input brings line B more positive than line A. At the RS-485 receiver, if input A is more positive than input B, the TTL output is logic high, and if input B is more positive than input A, the TTL output is logic low.

Referenced to the receiver's ground, each input must be within the range -7V to +12V. This range allows for differences in ground potential between the driver and receiver. The maximum allowed differential voltage, or the difference between the voltages on line A and line B, is ±6V.

Why Balanced Lines Are Quiet

Balanced lines are quiet because the two signal wires carry nearly equal but opposite currents. The currents reduce received noise because most noise is

Table 6-1: TIA-485 and TIA-422 are balanced interfaces. TIA-422 allows just one transmitter per line.

Specification	**TIA-422-B (RS-422)**	**TIA-485-A (RS-485)**
Cable length @90 kbps, max. (ft), approximate	4000	4000
Cable length @10 Mbps.,max. (ft), approximate	50	50
Data rate, max. (bps)	10M	10M
Differential output (minimum, V)	± 2	± 1.5
Differential output (maximum, V)	± 10	± 6
Receiver sensitivity (V)	± 0.2	± 0.2
Driver load, minimum (Ω)	100	60
Maximum number of drivers	1	32 unit loads
Maximum number of receivers	10	32 unit loads

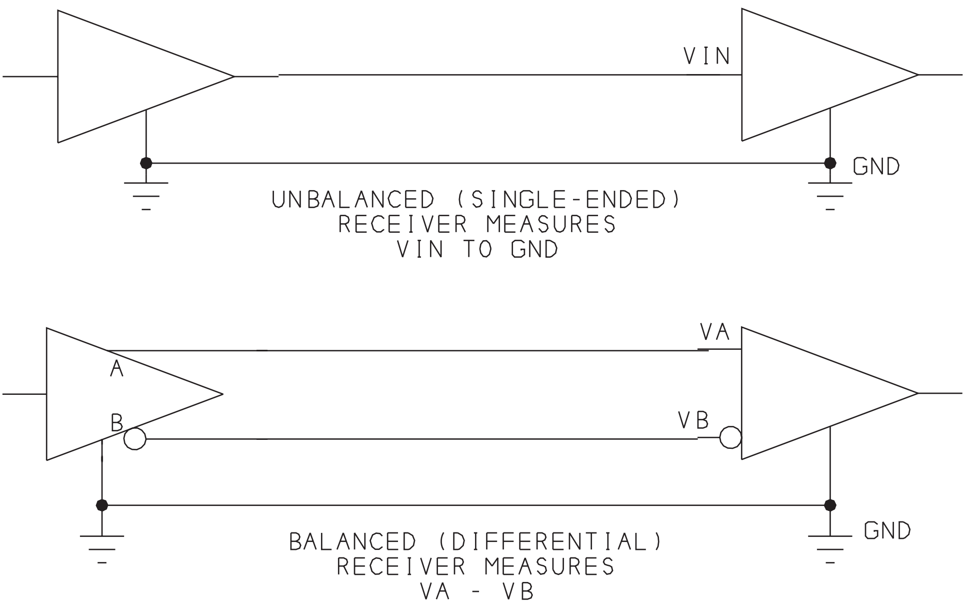


Figure 6-1: An unbalanced line uses one wire per signal while a balanced line uses two wires per signal. Both types of line must have a common ground reference.

present more or less equally on both wires, and any noise voltage that shows up on one line is canceled by an identical voltage on the other. The source of noise can be signals on other wires in the cable or signals that couple into the wires from outside the cable. A balanced receiver sees only the transmitted signal with noise eliminated or very much reduced.

In contrast, in an unbalanced interface, the receiver detects the voltage difference between the signal wire and ground. When multiple signals share a ground wire, each of the return currents induces voltages on the ground shared by all. Parallel interfaces can have eight or more lines switching constantly, and even serial interfaces often have two data lines and multiple status and control signals. If the ground connects to an earth ground, noise from other sources can affect the circuits as well.

Another advantage to balanced lines is that they are immune, within limits, to differences in ground potential between the driver and receiver. In a long cable, the grounds at the driver and receiver may vary by many volts. On an unbalanced line, ground differences can cause a receiver to misread an input. A balanced line can ignore mismatched grounds (up to a limit) because the receiver is detecting the difference between the two transmitted signals.

In reality, RS-485 components are guaranteed to withstand ground differences only up to the limit specified in their data sheets. A way to eliminate or reduce ground-voltage problems is to isolate the line so the driver's and receiver's ground potentials have no effect on the line. Chapter 7 shows ways to ensure that a line's ground potentials are within acceptable limits.

The Circuits Inside

Figure 6-2 shows the circuits inside an RS-485 driver and receiver. The components shown are the same as the equivalent circuits in the data sheet for the Texas Instruments SN75179B. Other RS-485 chips may differ in the details, but the overall operation is much the same.

The schematic shows the drivers' outputs and the receivers' input and output circuits along with the path current takes when the line transmits a TTL logic high. Not shown are the circuits between the driver's TTL inputs and the output transistors and between the RS-485 receiver circuits and the TTL outputs.

A logic high at the driver's TTL input causes transistors Q1 and Q4 to switch on and Q2 and Q3 to switch off. The voltage on line A causes Q6 to switch on. Current flows into Q6 and returns to the driver via the ground wire. In a similar way, the low voltage on line B causes Q7 to switch on, and current flows

from Q7 into Q4, returning to the receiver via the ground wire. Line A is more positive than line B, and the result is a logic high at the receiver's TTL output.

Each driver's current forms a complete loop from driver to receiver, then back to the driver. A ground wire or other ground connection provides a return path for both signals. Because the two ground currents are equal and opposite, they cancel each other, and the actual current in the ground wire is near zero.

If the line has multiple receivers, each behaves like the one shown. If the line has termination resistors, current flows in these as well.

For a logic low, the situation is the reverse. Q2, Q3, Q5, and Q8 switch on, the other transistors switch off, and the current in the wires flows in the reverse direction.

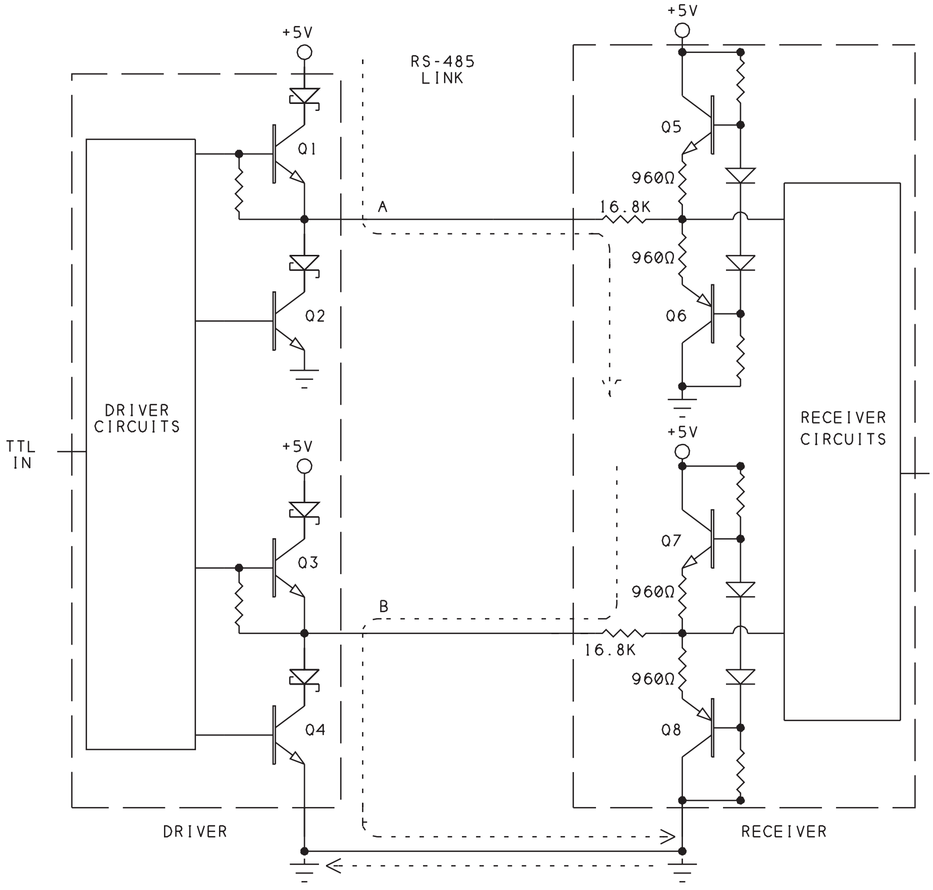


Figure 6-2: The circuits inside an RS-485 driver and receiver.

Voltage Requirements

RS-485 interfaces typically use a single power supply of 3.3V or 5V, but the logic levels at the drivers and receivers differ from the voltages used by 3.3/5V TTL/CMOS logic chips. For a valid output, the difference between outputs A and B must be at least 1.5V.

Figure 6-3 shows an RS-485 driver's A and B pins when functioning as outputs, each referenced to signal ground. The driver's power supply is +5V. When A = +4V, and B = +1V, the differential output (A - B) is +3V. When A = +1V and B = +4V, the differential output is -3V.

If one output switches before the other, the combined differential output switches more slowly, limiting the maximum bit rate of the line. Skew is the time difference between the two outputs' switching. RS-485 drivers are designed to minimize skew. For example, Linear Technology's LTC1685 guarantees a maximum skew of ±3.5 ns.

At the RS-485 receiver, the difference between A and B needs to be just 0.2V. If A is at least 0.2V more positive than B, the receiver's output is a logic high, and if B is at least 0.2V more positive than A, the receiver's output is a logic low. If the difference between A and B is less than 0.2V, the logic level is undefined.

The difference between the requirements at the driver and receiver results in a noise margin of 1.3V. The differential signal can attenuate or have noise spikes as large as 1.3V, and the receiver will still see the correct logic level. The noise

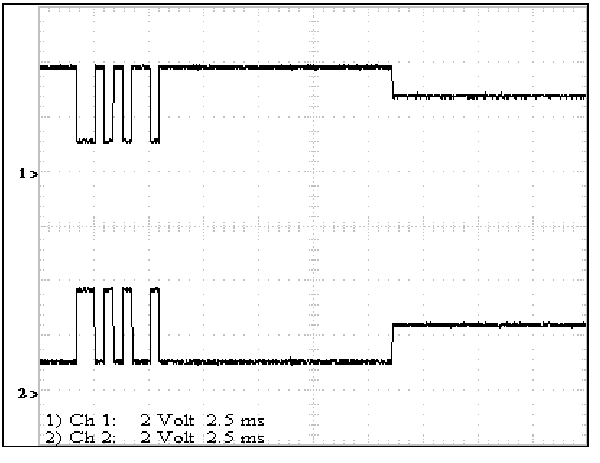


Figure 6-3: An RS-485 driver's outputs, referenced to ground. Line B (bottom) is the inverse of line A (top).

margin is less than on an RS-232 line, but RS-485's differential signals cancel most noise to begin with. Also, on most lines, the differential voltage is larger than the minimum 1.5V, so the noise margin is larger. A driver powered at 3.3V can easily provide outputs that differ by 1.5V.

TIA-485-A defines logic 1 as the state where B > A and logic 0 as the state where A > B. In RS-485 interface chips, a logic-high TTL input results in A > B, and a logic-low TTL input results in B > A. In virtually every UART, a logic-high voltage corresponds to a logic-1 data bit (or a Stop bit or idle line), and a logic-low voltage corresponds to a logic-0 data bit (or Start bit).

Using these definitions, a logic 0 at a driver's input results in a logic 1 on the RS-485 line and a logic 0 at the receiver's output. In reality, the logic-level definitions don't matter as long as all of the computers agree on a convention.

Connecting a transceiver's A pin to the cable's B line and connecting the B pin to the A line has the effect of inverting the signals. But swapping the lines to invert the signal can cause trouble when you return later to modify or debug the link and have long forgotten that you deliberately swapped the lines. Also, the chips' internal (and possibly external) biasing brings A high and B low, and swapping the lines conflicts with these biasing circuits.

Current and Power

The total current used by an RS-485 network varies with the impedances of the components including drivers, cable, receivers, and termination components. A low output impedance at the driver and a low-impedance cable enable fast switching and ensure that the receiver sees the largest signal possible. A high impedance at the receiver decreases the current on the line and thus reduces power consumption.

When used, termination components can greatly increase the current a line consumes. Many RS-485 lines have a 120Ω resistor across the differential lines at each end of the line. The parallel combination of the resistors is 60Ω. The terminations create a low-resistance path from the driver with a logic-high output, through the terminations, and into the driver with a logic-low output. On short cables at slow bit rates, you may be able to eliminate the termination entirely and greatly reduce power consumption.

With no termination components, the receivers' input impedance has the greatest effect on the series resistance of the line. The total input impedance varies with the number of enabled receivers and their input impedances. Chapter 7

explains how to decide whether to use a termination and if so, what components to use..

In addition to a 60Ω parallel termination, an RS-485 driver can drive receivers that draw up to a total of 32 unit loads. TIA-485-A defines a unit load in terms of current consumption. A receiver equal to one unit load draws no more than a specified amount of current at input-voltage extremes specified by the standard. When the received voltage is as much as +12V greater than the receiver's signal ground, a receiver equal to 1 unit load consumes no more than 1 mA. When the received voltage is as much as 7V less than the receiver's ground, the same receiver consumes no more than -0.8 mA. To meet this requirement, a receiver must have an input resistance of at least 12k between each differential input and the supply voltage or ground, depending on the direction of current flow.

The resistance at each of the two differential inputs of a 1-unit-load receiver is thus 12k. (This is the resistance from an input to ground or the supply voltage, not the resistance between the two inputs.) Add a second receiver, and the parallel resistance of the combination drops to 6k. With receivers equivalent to 32 unit loads, the parallel resistance of the combined inputs is just 375Ω, or slightly less due to leakage currents.

This routine obtains the parallel input resistance for networks with receivers equivalent to 2 to 32 unit loads:

VB

```
Public Sub GetParallelInputResistance()

    Const receiverInputResistance = 12000

    Dim parallelInputResistance As Single = receiverInputResistance

    For numberOfUnitLoads As Integer = 2 To 32

        parallelInputResistance = _
            ((receiverInputResistance * parallelInputResistance) / _
            (receiverInputResistance + parallelInputResistance))

        Console.Write("With  " & numberOfUnitLoads)
        Console.Write(" unit loads, parallel input resistance = ")
        Console.WriteLine(Math.Round(parallelInputResistance))

    Next numberOfUnitLoads
End Sub
```

VC#
```
public void GetParallelInputResistance()
{
        const Int32 receiverInputResistance = 12000;

        Single parallelInputResistance = receiverInputResistance;

        for (Int32 numberOfUnitLoads = 2; numberOfUnitLoads <= 32;
                numberOfUnitLoads++ )
        {
                parallelInputResistance =
                        ((receiverInputResistance * parallelInputResistance) /
                        (receiverInputResistance + parallelInputResistance));

                Console.Write("With  " + numberOfUnitLoads);
                Console.Write(" unit loads, parallel input resistance = ");
                Console.WriteLine(Math.Round(parallelInputResistance));
        }
}
```

Receivers that are a fraction of a unit load have higher input resistances. For example, the input resistance of a 1/8-unit-load receiver is 96k, and the total parallel resistance of 32 of these receivers is 3k. A receiver that is a fraction of a unit load may be slower than other receivers though some, such as the 1/8-unit-load MAX3088, can operate at up to 10 Mbps.

A line can have more than 32 unit loads if the network's common-mode voltage range is less than the maximum allowed or if the line's differential load is greater than 60Ω. Chapter 7 has more about common-mode voltages. Termination components can reduce the maximum allowed number of unit loads.

Speed

An RS-485 line can have a bit rate as fast as 10 Mbps or as long as 4000 ft but not both at the same time. Longer cables require slower bit rates because the cable's capacitance slows the signal transitions. Figure 6-4 is a general guideline for determining allowed bit rate for a cable length as recommended by TIA-422.

At rates of up to 90 kbps, RS-485 and RS-422 support cable lengths of up to 1200 meters (4000 ft). At faster rates, the maximum allowed cable length drops, to around 120 meters (400 ft) at 1 Mbps and 15 meters (50 ft) at 10 Mbps. The graph assumes an unshielded, terminated twisted pair with a wire gauge (AWG) of 24.



Figure 6-4: RS-485 supports transmissions up to 10 Mbps, but the higher bit rates require shorter cables.

Internal Protection Circuits

In a half-duplex line, only one driver should be enabled at a time. But no matter how carefully a network is designed, if there are multiple drivers, there's a chance that two or more drivers will be enabled at once.

When two or more drivers are enabled and try to pull a line to opposite states, the result is unpredictable voltages and high currents. Figure 6-5 illustrates. When Output 1 is a logic high, the signal line has a low impedance to the supply voltage. If Output 2 switches on and is a logic high, there is no problem. But a logic-low output has low impedance to ground. The result is a low-impedance path from the power supply, through Output 1 and Output 2, to ground. The components draw high currents, and the voltage is likely to be an undefined logic level. This situation is called line contention.

All RS-485 interface chips include current limiting and thermal shutdown to protect the chips if more than one driver is enabled at once. TIA-485-A

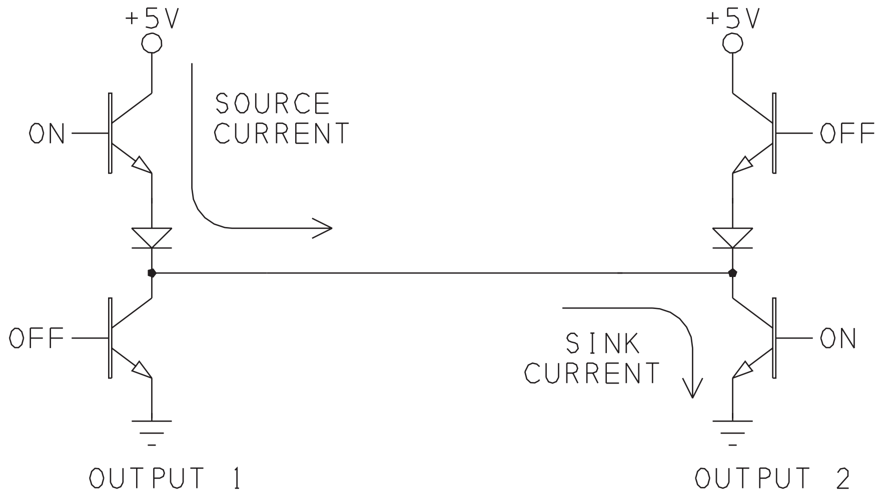


Figure 6-5: When two or more outputs are on at the same time, the resulting low impedance path from +5V to ground draws high currents and makes the output voltage unpredictable.

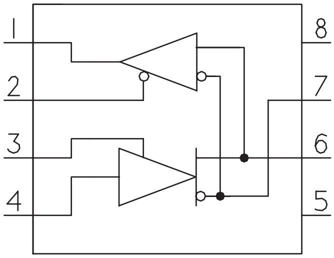
requires limiting the current to 250 mA. If an output continues to source or sink high currents, the chip heats up, and eventually the chip's thermal shutdown circuits switch the output to a high-impedance state. Of course, this makes the output unusable until it cools down, but at least the components survive.

Interfacing Options

Interface chips are available to convert between TTL/CMOS and RS-485 logic levels. RS-232 interfaces can also be converted to RS-485. Converter modules are available from a variety of sources including B&B Electronics Manufacturing Company (*www.bb-elec.com*) and R.E. Smith (*www.rs485.com*).

Chips

Sources for RS-485 interface chips include Intersil Corporation, Linear Technology, Maxim Integrated Products, National Semiconductor, Sipex Corporation, Texas Instruments, and Zywyn Corporation. Chips are available with just about any combination of features and abilities you might want. Check manufacturers' websites for the latest offerings.



SN75176B
RS-485
TRANSCEIVER

PIN	NAME	DESCRIPTION
1	RO	RECEIVER OUTPUT
2	$\overline{RE}$	RECEIVER ENABLE
3	DE	DRIVER ENABLE
4	DI	DRIVER INPUT
5	GND	GROUND
6	A	RS-485 LINE A
7	B	RS-485 LINE B
8	VCC	POWER (+5V)

Figure 6-6: Many half-duplex RS-485 interface chips use the same configuration and pinout as the SN75176B.

Figure 6-6 shows a chip that contains a single RS-485 transceiver. A driver converts a TTL/CMOS voltage at DI to an RS-485 differential voltage, and a receiver converts an RS-485 differential voltage to a TTL/CMOS-compatible voltage at RO. The driver and receiver each have an enable input. The Texas Instruments SN75176B is an early chip that used this configuration and pinout. Dozens of other chips use the same pinout, often with different performance or added features. For example, the MAX3485 uses a 3.3V supply.

Some chips have maximum bit rates lower than RS-485's 10 Mbps maximum. As Chapter 7 explains, these chips can provide better signal quality in lines that don't require fast bit rates. The MAX3082 has a maximum of just 115,200 bps. Other chips and their maximum bit rates include the MAX483 at 250 kbps, the MAX481 at 2.5 Mbps, and National Semiconductor's DS16F95 at 5 Mbps. Linear's LTC1685 supports speeds of up to 52 Mbps.

The MAX1483 has receivers that are 1/8 unit load each, allowing up to 256 transceivers on a bus. To save power, the MAX481 and other chips enter a low-power shutdown mode when both the driver and receiver are disabled.

Many chips include features to protect the circuits from damage due to high voltages or ESD. The MAX13444E has ±80V fault protection and allows hot swapping. Texas Instruments' SN65LBC184 has transient protection, a receiver that is 1/4 unit load, and an extended temperature range. The MAX1480A is a complete isolated interface on a chip.

The MAX3085 and other chips include fail-safe circuits that guarantee a logic-high receiver output when the receiver's inputs are open or shorted. Linear's LTC2859 has an internal 120Ω termination controlled by a termination-enable input.

Adding a Port on a PC

As with RS-232, you can add an RS-485 port to a PC by installing an expansion card or PC Card or attaching a USB converter. You can also convert an RS-232 or TTL serial port to RS-485. RS-485 ports typically appear as COM ports in PCs. Some cards and converters have ports with one pair of data lines for half-duplex communications, while others have two pairs for full duplex communications. Some half-duplex ports include hardware support for automatically enabling the driver when needed.

Converting 3.3/5V Logic

The asynchronous serial port on a microcontroller can use 3.3V CMOS or 5V TTL/CMOS logic levels. There are several ways to convert between 3.3/5V logic and RS-485.

Full Duplex

Most RS-485 lines are half-duplex, where multiple drivers and receivers share a signal path. But you can also use RS-485 in a full-duplex line, where each direction has its own signal path. As long as you include any required flow-control signals in the interface, you can swap an RS-232 line for a full-duplex RS-485 with no changes to the software or firmware that uses the interface. Both can use the same programming, though RS-485 supports higher bit rates and the hardware allows longer cables.

For a full-duplex line, you can use the Texas Instruments SN75179B or similar chips. The SN75179B contains a driver that translates 5V TTL signals to RS-485 and a receiver that translates RS-485 back to 5V TTL. Figure 6-7 illustrates.

This chip is a solution when you want to create a long-distance, full-duplex link between microcontrollers. The RS-485 interface chips are smaller and cheaper than RS-232 interface chips. You can also use RS-422 interface chips for this type of link.

Figure 6-8 shows how to use multiple drivers and receivers in full-duplex communications. One arrangement is in a primary/secondary network, where a primary computer (Node 0 in the figure) has control of the network and grants the secondary computers (Nodes 1 and higher) permission to transmit. One pair of wires connects Node 0's driver to all of the secondary nodes' receivers. In the other direction, another pair of wires connects all of the secondary nodes'

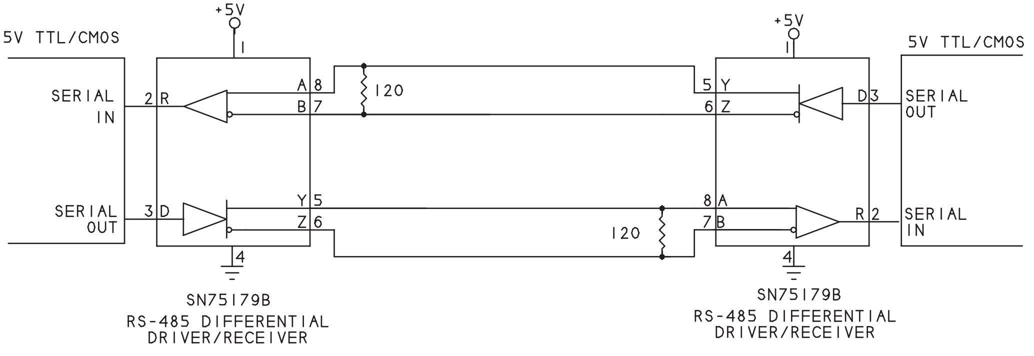


Figure 6-7: A full-duplex RS-485 circuit provides a data path for each direction..

drivers to Node 0's receiver. All of the secondary nodes monitor messages from Node 0. The node being addressed replies on the other pair of wires. The advantage is that the secondary nodes don't see the other nodes' replies. With a single data path, all of the secondary nodes receive all of the network traffic.

Half Duplex

Many RS-485 circuits are half-duplex, where multiple drivers and receivers share a signal path. The interfaces form a serial network, and each computer with an RS-485 interface is a node in the network.

Networks typically use half duplex interfaces and allow one node at a time to transmit. Links with just two devices can be half duplex as well. Microcontrollers that allow configuring a port bit as input or output can send and receive on a single port bit, reconfiguring the bit as needed. You might use this approach if you need to use the fewest number of port pins or wires possible. If you need to transmit in just one direction (simplex), you of course need only one data path.

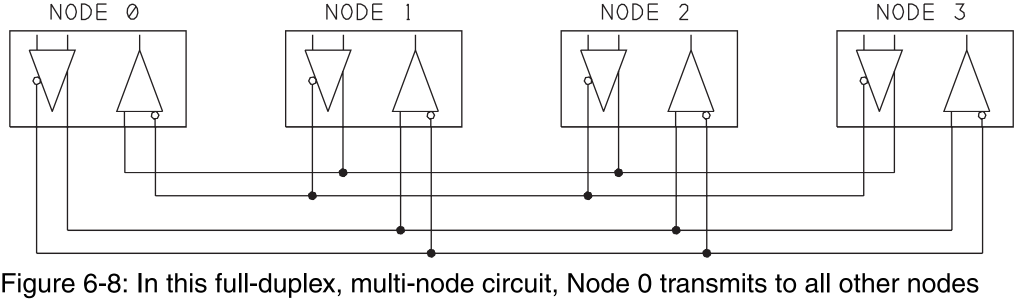


Figure 6-8: In this full-duplex, multi-node circuit, Node 0 transmits to all other nodes on one line and receives from all other nodes on the other line.

Figure 6-9 shows a half-duplex network that uses an SN75176B differential bus transceiver.

When a driver's enable input (DE) is low, the driver's output is high impedance, and for all practical purposes the driver is removed from the circuit. When a receiver's enable input (/RE) is high, the receiver's output is high impedance and no longer follows the RS-485 input. Chapter 7 has more about biasing this type of network to ensure valid signal levels when the network lines are idle, open, or shorted.

Converting RS-232

Some applications require converting RS-232 signals to RS-485. If a computer has an available RS-232 port, adding an external converter can be cheaper and easier than buying and installing an RS-485 card or USB converter. Converter modules are available from a variety of sources, or you can make your own. If you need high speed, be aware that RS-232 drivers can limit the top speed of the RS-485 interface.

Microcontroller development boards sometimes have RS-232 interfaces built-in. If you need RS-485, it might be easier to bypass the RS-232 interface by removing the RS-232 interface chip or the connections to it and wiring the RS-485 interface directly to the microcontroller's port pins.

Half Duplex

Figure 6-10 shows one way to convert RS-232 to RS-485. The interface uses three RS-232 lines: TX transmits data, RX receives data, and RTS controls direction. A MAX233 converts the RS-232 signals to TTL levels, and the TTL signals connect to an SN75176B or similar chip that provides the RS-485 interface.

When RTS is low, the enable inputs of the SN75176B are high and TX can transmit on the RS-485 line. When RTS is high, the enable inputs are low and RX can receive data on the RS-485 line. A circuit can use DTR instead of RTS.

It's likely that the cable from the RS-232 port to the converter will be no more than a few feet long. In this case, you can use Chapter 4's short-range transistor circuit in place of the MAX233 or other converter chip.

In a similar way, you can create full-duplex RS-232-to-RS-485 interfaces using a SN75179B or other 4-wire RS-485 chip.

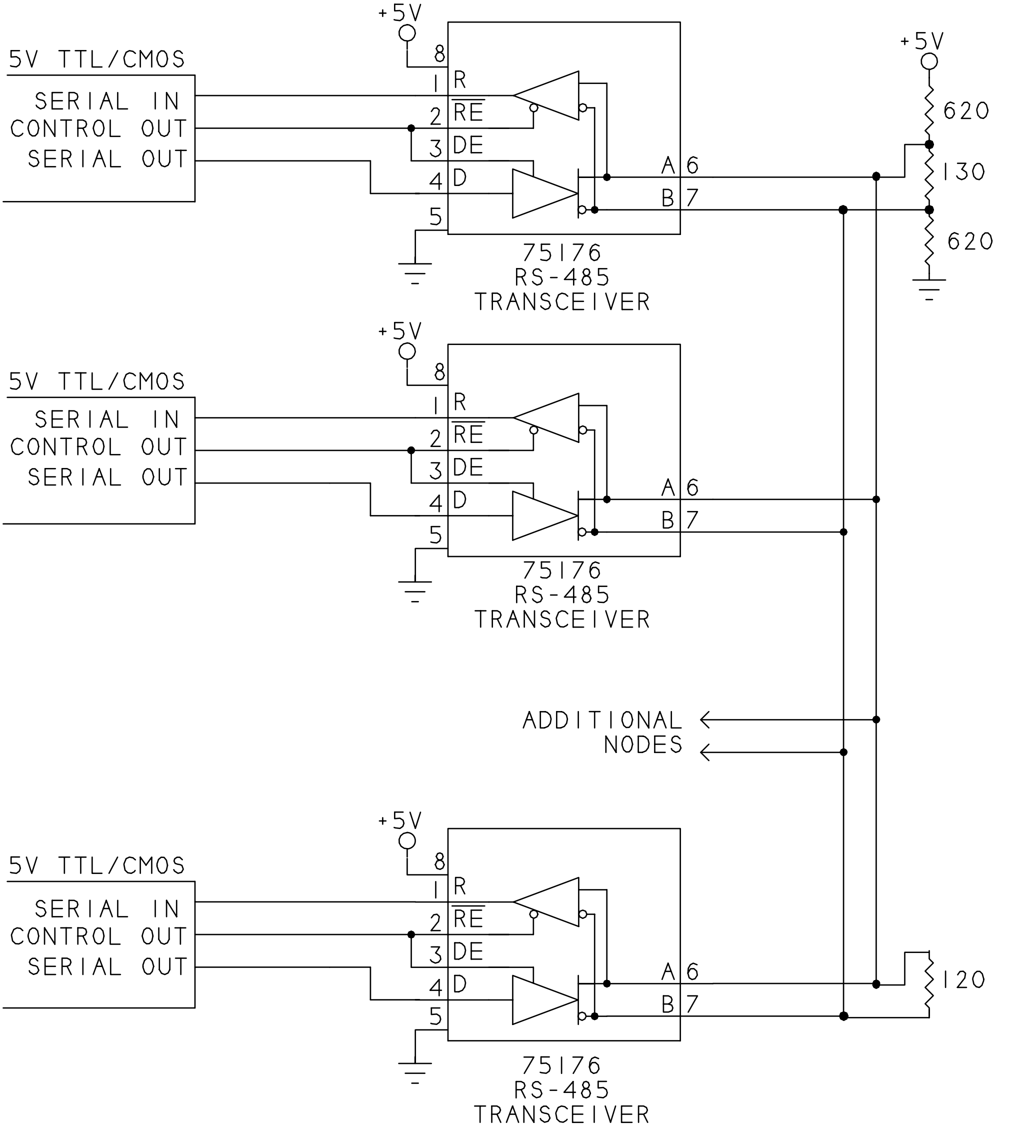


Figure 6-9: This half-duplex RS-485 network has a single data path. All nodes must also share a gorund connection, typically via a ground wire in the network cable..

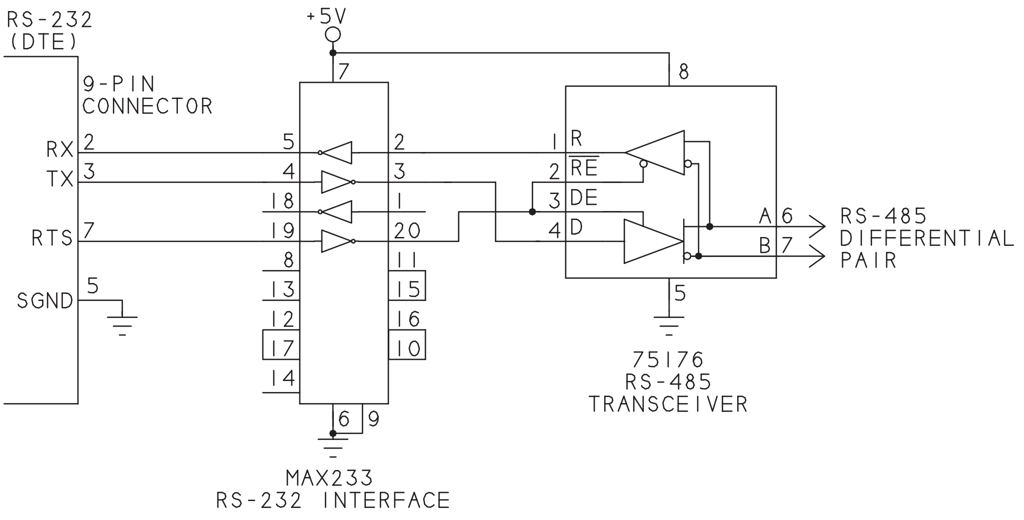


Figure 6-10: This circuit converts between RS-232 and TTL, and between TTL and RS-485.

A single-chip solution is the MAX3162E, which contains two RS-232 drivers, two RS-232 receivers, and an independent RS-485 transceiver. If you're designing a circuit that might use either interface, the MAX3160E can interface 5V logic to either RS-232 or RS-485 as selected by the Mode Functionality input pin.

Full Duplex

Sometimes when you have two devices with different interfaces, all you want is to link them as cheaply and simply as possible. Over short distances, you can connect a full-duplex RS-485 or RS-422 interface to RS-232 with a few inexpensive components.

Figure 6-11 shows a connection between an RS-232 port and an RS-485 port. RS-485's B output connects to RS-232's RX input. Referenced to signal ground, the B output is near 0V for a logic 1 and near +5V for a logic 0. As explained in Chapter 4, these voltages don't meet RS-232's minimum specification, but RS-232 receivers typically use 5V TTL thresholds. RS-485's A output is unused and left open.

In the other direction, a voltage divider ensures that the differential input voltage doesn't exceed RS-485's maximum of ±6V. The receiver's A input is tied to ground. The B input sees about 1/3 of the transmitted RS-232 voltage. If the

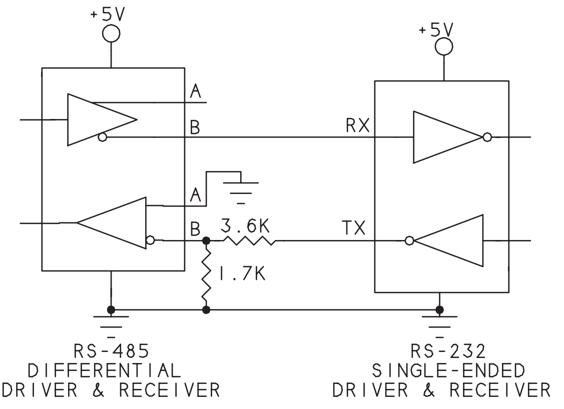


Figure 6-11: Use this circuit for a short link between a device with an RS-485 interface and one with an RS-232 interface.

RS-232's TX is ±15V, the RS-485 receiver's input sees a voltage of about ±5V. If the RS-232's TX is just ±5V, the RS-485 receiver's input sees a differential voltage of about ±1.6V, which is well above the minimum requirement of ±0.2V.

This interface requires a full-duplex RS-485 or RS-422 interface. If the differential receiver can accept input voltages as large as the RS-232 driver's outputs, you don't need the voltage divider and can connect the driver and receiver directly.

Both interfaces invert the signals, so a TTL logic 1 at one end translates to a TTL logic 1 at the other end. This circuit is usable over short distances at slow bit rates. If you don't need to invert the signals, use RS-485's line A and connect line B to ground.

Controlling the Driver Enable

In an RS-485 network, only one driver can be enabled at a time. After sending data, a node should disable its driver as quickly as possible so the next node to transmit can enable its driver and start transmitting.

Figure 6-12 shows a transmitted byte and its driver-enable signal. A node can control the driver-enable line with software or with a hardware-only method.

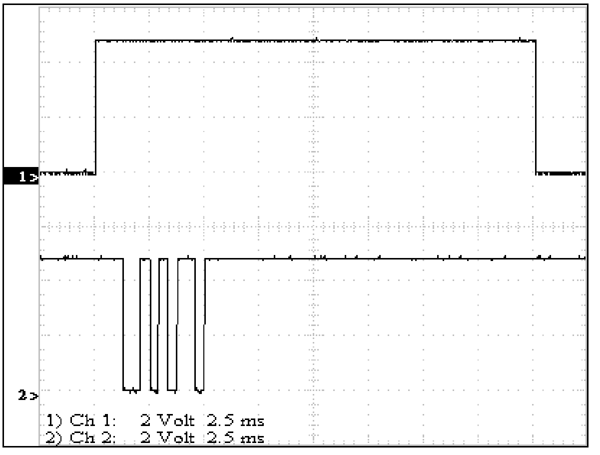


Figure 6-12: Trace 1 is the active-high driver-enable signal. Trace 2 is a transmitted byte. The driver-enable must go high before the byte transmits, and can return low any time after the transmission has completed.

Re-enabling the Driver

If a transmitting computer disables its driver by the middle of the final transmitted Stop bit, the next computer to transmit can enable its driver almost immediately after detecting the Stop bit.

For some of the driver-control methods described below, the driver may remain enabled for a time after the final Stop bit. In these cases, the next node to transmit must be sure to wait long enough to ensure that the previous computer has had enough time to disable its driver.

In many applications where a computer transmits and then expects a response, the required delay has elapsed by the time the responding computer has prepared the data to send. When needed, a way to ensure a delay is to start a timer after receiving data and then transmit after the timer times out. The amount of time to wait varies with the method the transmitting computer uses to control its driver-enable line.

Software-assisted Control

Controlling the driver in software requires a dedicated output bit that connects to the transceiver's driver-enable input. On a PC's RS-485 port that uses an RS-232 converter or USB/serial converter, RTS or DTR can serve as the driver-enable output. Other ports can use any spare output bit.

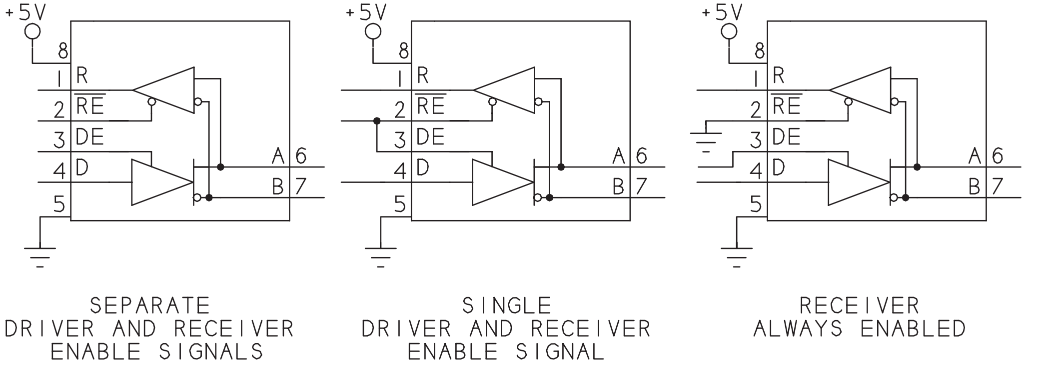


Figure 6-13: Three options for controlling the driver-enable and receiver-enable inputs on an RS-485 transceiver.

Configurations

An RS-485 port can use any of three configurations (Figure 6-13):

- Use two output bits and control the driver's and receiver's enable inputs separately. This arrangement provides the most flexible control but requires two port bits.
- Use one output to control both enable lines. The driver's enable input is active-high and the receiver's enable input is active-low so either the driver or receiver is always enabled and both are never enabled at the same time. This configuration is useful if a node doesn't need to receive its own transmissions.
- Control the driver-enable input only. Tie the receiver's enable input low to keep the receiver enabled at all times. With this configuration, a node receives the data it sends and thus can verify that the data transmitted.

Disabling the Driver

When using software control, the transmitting node must disable the driver-enable line as soon as possible after transmitting all of the data. To determine when to disable the driver, the software can wait for the transmit buffer to be empty, read back all transmitted data, or calculate a delay time.

Wait for the Transmit Buffer to Empty

An indication that the transmit buffer is empty can be useful in detecting when it's safe to disable the driver.

In the PIC18F4520 and other PIC® microcontrollers, the TXSTA register's TRMT bit indicates whether the transmit shift register is empty or full. Chapter 11 has more about PIC18F4520 programming.

In PCs, .NET's SerialPort class includes the BytesToWrite property, which returns the number of bytes in the transmit buffer. However, BytesToWrite returns zero when the UART's hardware buffer still contains bytes to transmit and when the final byte is transmitting. If you use BytesToWrite to find out if the data has transmitted, you'll need to add a delay after BytresToWrite = 0 to allow the UART's buffer to empty. Chapter 9 and Chapter 10 have more about .NET programming.

Read Back the Transmitted Data

A transmitting computer can ensure that all data has transmitted by reading the data back after it transmits. This method also detects problems such as a disconnected transceiver or multiple drivers enabled at the same time. To enable reading back the data, the RS-485 receiver must be enabled when transmitting.

To use this method, the program code sends the data and read back the data just transmitted. To verify a match, the transmitting computer can just count the received bytes or compare the received data with the bytes transmitted. On receiving the expected data or number of bytes, the transmitting computer can disable its driver. If the data doesn't appear or the values don't match, the transmitting computer can retry or give up.

Calculate a Delay

A calculated delay is another approach to determining when to disable the drivers. Program code uses the bit rate and number of bytes to calculate the time required to send the data. The code enables the driver, sends the data and waits the calculated time before disabling the driver. If using flow control, the time required to transmit a block of data might be impossible to calculate. A solution is to calculate and wait a 1-byte delay time just after the final byte of data transmits.

Hardware Control

Hardware methods of controlling the driver-enable line can be efficient and eliminate the need for dedicating a port bit and program code to control the bit.

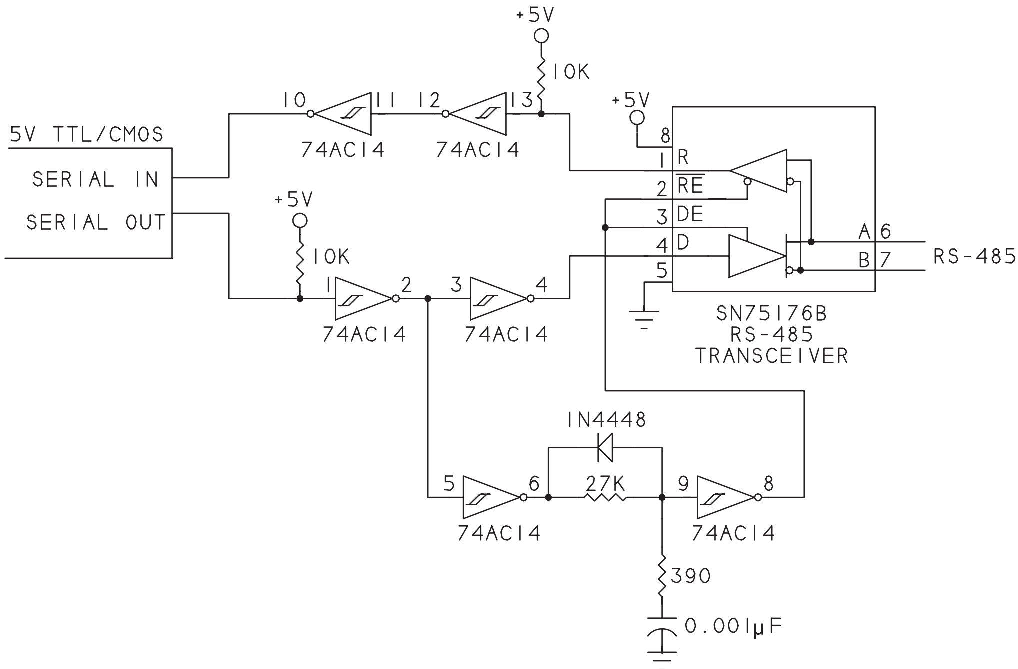


Figure 6-14: This circuit controls the driver-enable line entirely in hardware. Circuit courtesy of R.E. Smith (www.rs485.com).)

Deriving the Driver Enable from the Data

The ideal driver-enable control requires no software control, no user configuring, and no delays between receiving and transmitting data. Figure 6-14 shows a circuit that meets these goals.

The added components add some complexity to the circuit design. However, the components are inexpensive and the freedom of not having to determine when it's OK to enable and disable the driver improves network performance and can be well worth the cost.

Chapter 7 described the two valid states of an RS-485 line. When input A is at least 200 mV more positive than input B, the line may be idle or transmitting a Stop bit or a logic-1 data bit. When input B is at least 200 mV more positive than input A, the line may be transmitting a Start bit or a logic-0 data bit.

If a transmitting computer's driver is disabled by the end of the final Stop bit, another computer in the network can enable its driver and send a Start bit right away. There's no need to delay to be sure the previous driver has been disabled.

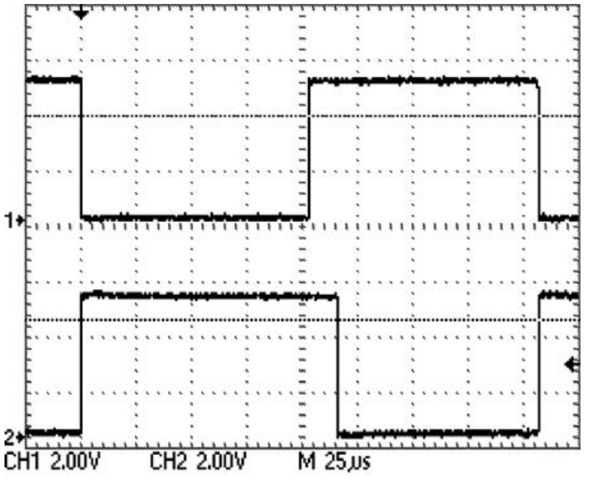


Figure 6-15: A transceiver's driver-enable signal (bottom) can be derived from the transmitted data (top).

One way to ensure that a driver is disabled by the end of the Stop bit is to derive the driver-enable signal from the data. The driver-enable signal can go low to disable the driver during every Stop bit, logic-1 data bit, and whenever the driver is idle (isn't transmitting a word). The driver-enable signal can go high to enable the driver during every Start bit and logic-0 data bit. Figure 6-14's circuit uses this approach. An added benefit of the circuit is reduced power consumption. No drivers are enabled when the line is idle and a driver is enabled for only an average of half the time required to transmit a word.

At RS-485 transceiver SN75176B, when the data input at pin 4 goes low (logic 0), the driver-enable input (DE) at pin 3 goes high, enabling the driver and causing the differential data on pins 6 and 7 to follow pin 4. When pin 4 goes high (logic 1), pin 6 of Schmitt-trigger inverter 74AC14 goes high. The 0.001µF capacitor charges through the 27k and 390Ω resistors. After a short delay, pin 9 of the 74AC14 goes high and DE goes low. DE thus remains high while the line switches to logic 1 and for a short time afterwards.

After the delay, pin 3 on the SN75176B goes low, disabling the driver. The data on pins 6 and 7 follows pin 4 as it switches state and remains at a logic 1 after the driver is disabled (Figure 6-15). When the data input goes low again, DE follows, going high to enable the driver.

With the components shown, the delay in switching the driver off is approximately 27 µs. Ideally, the delay is half of one bit width. In reality, the computers

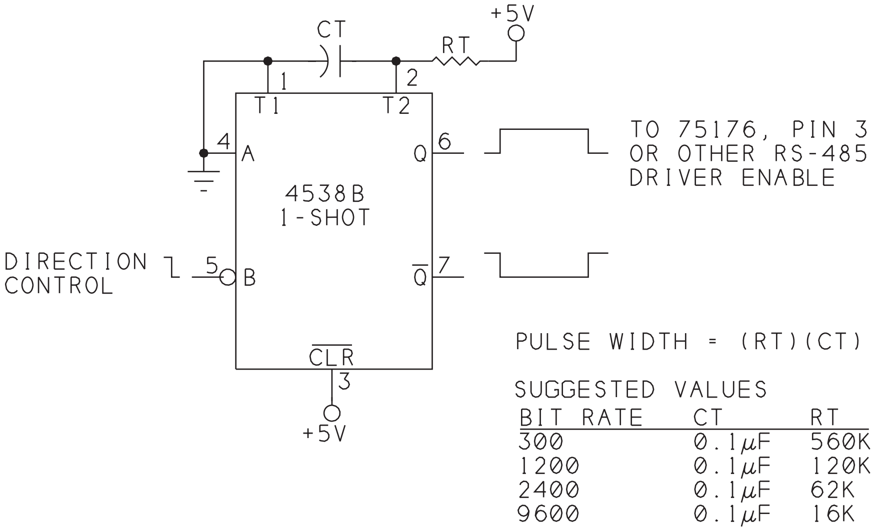


Figure 6-16: This retriggerable 1-shot can automatically enable an RS-485 driver. A Start bit or logic 0 results in a pulse of 1.5 word widths.

in most networks don't respond immediately after the Stop bit so a slightly longer delay causes no harm.

To reduce the delay, decrease the value of the 27k resistor. Each kilohm of resistance corresponds to approximately 1 µs of delay.

The circuit has a small effect on the transmitted RS-485 data. When the data input goes low, the driver's output switches from disabled to logic 0. When the data input goes high, the driver's output is already enabled and switches from logic 0 to logic 1. At very fast bit rates or when the bit rates of the transmitting and receiving computers don't closely match, differing delays in switching the driver on and switching the output of an enabled driver can result in errors when reading data. The transceiver's data sheet specifies the maximum delays for a specific load. The relevant delays are driver-enable to output and driver input to output. At low to moderate bit rates, the asymmetry is small enough that it causes no problems.

Using a One Shot

For applications that can't use the approach described above, Figure 6-16 shows an alternative. The circuit uses a retriggerable 1-shot multivibrator to enable the driver whenever the node is transmitting.

The 1-shot uses the Start bit to detect when a byte begins to transmit. Each byte begins with a falling edge that signifies the Start bit. Capacitor CT and resistor RT set the width of the 1-shot's output pulse to slightly longer than the time to transmit one byte.

On the falling edge of the Start bit, the one-shot's output goes high for at least one delay time. Because the 1-shot is retriggerable, if another byte begins to transmit before the 1-shot times out, the new byte's Start bit holds the output high until that byte has finished transmitting. If another byte doesn't arrive, soon after the last byte's transmission is complete, the one-shot's output goes low and the driver is disabled.

Because the 1-shot retriggers on every falling edge, the time that its output remains high varies depending on the data sent. If bit 6 = 1 and bit 7 = 0, the falling edge at bit 7's transition to zero will retrigger the 1-shot and result in a longer pulse. (Remember that the data transmits least-significant-bit first.) But if the byte is all 1s or all 0s, the 1-shot triggers only on the falling edge of the Start bit, and the pulse is shorter.

Otherwise, a jumper or switch can match the delay time to the bit rate. Components selected to work at the slowest expected bit rate will result in much longer delays than needed at faster bit rates.

The 1-shot's delay isn't precise, so you'll need to allow a margin of error when selecting components. The driver thus is likely to remain enabled longer than needed, and the node that transmits next must delay before enabling its driver.

USB Controller with Hardware Driver Enable

USB/RS-485 converters can use FTDI Chip's FT232R USB UART. This USB device controller has an asynchronous serial interface and an output that is high while data is transmitting. Chapter 15 has more about this chip.

This page intentionally left blank

7

Designing RS-485 Links and Networks

In RS-485 communications, the choice of cables and components can mean the difference between systems that communicate flawlessly every time and systems that fail either immediately or at seemingly random and thus unpredictable times.

The following guidelines will help in designing RS-485 circuits that work without problems:

1. Use the slowest drivers possible for the bit rate.
2. Terminate long lines with their characteristic impedance.
3. Wire the nodes in a bus topology.
4. Bias inactive links.
5. Use twisted-pair cable.
6. Limit common-mode voltages.

This chapter explains the reasons behind the guidelines and shows how to follow the guidelines in your designs.

Long and Short Lines

RS-485 can connect anywhere from 2 to over 200 computers. The bit rate can be as slow as 300 bps or less on up to 10 Mbps. Transceivers that support even higher rates are available for applications that don't need to comply with TIA-485-A. Cables can be very short or thousands of feet long.

Over short distances at slow bit rates, the component and cable choices are less critical, though even here the right choices can save power and reduce noise. Over long distances and at fast bit rates, selecting the proper cables, drivers, receivers, and related components is critical.

When Is a Line Long?

The theory behind how digital signals behave in long-distance lines requires thinking about the voltages on the line in a different way. On a long line, you can't assume that a voltage transfers instantly and perfectly from driver to receiver.

An RS-485 cable can behave as an electrically long or short line. The terms long and short don't refer to the cable's physical length but instead to the amount of time required for a signal to propagate along a wire to the receiver. The amount of time varies with the physical length of the wires, the frequencies in the signals carried by the wires, and how fast the signals travel in the wires.

When the wires are physically short and the frequencies are low, the time required for signals to propagate down the wires has little effect on signal quality. The circuit is considered a lumped system, and the wires form a short line. In many respects, you can think of short lines as perfect, zero-impedance conductors. When an output switches state, you can assume that the input at the other end of the cable instantly sees an identical voltage.

When the wires are physically long and the frequencies are high, the time required for a signal to propagate down the wires is significant. This type of circuit is considered a distributed system, and the wires form a long line. Another name for a long line is transmission line. On a long line, terminating components can help ensure that the receiver sees a clean signal by reducing reflected voltages on the line.

Understanding how long and short lines behave requires understanding the effects of two parameters: rise time and cable delay.

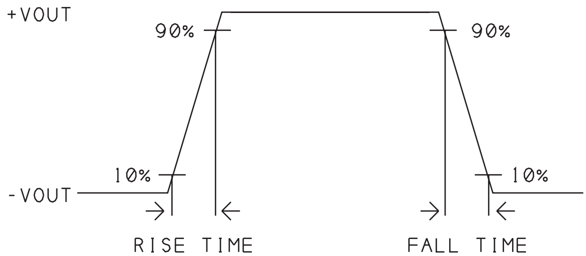


Figure 7-1: A circuit with short rise and fall times contains high frequencies, even if the bit rate is slow..

Rise Time

Rise time is the time required for an output to switch from 10% to 90% of full range (Figure 7-1). Fall time is the time required for an output to switch from 90% to 10% of full range. The rise and fall times are often equal or nearly equal, so some sources use rise time to refer to transition time in general. The rise time of a digital signal is an indication of the frequencies that make up the signal. A faster rise time means the signal contains higher frequencies.

The data sheets for RS-485 drivers specify typical and maximum rise and fall times. The values range from a few nanoseconds to nearly a microsecond. The specifications assume a defined load, often 54Ω and 50 or 100 pF, with higher capacitance resulting in a longer switching time.

A shorter rise time corresponds to a faster slew rate and faster possible bit rates. In general, the time required to transmit a bit should be 5–10 times longer than the rise time to ensure that the voltage has reached a valid logic level by the time the receiver reads the bit. For example, an RS-485 driver rated for use at 2.5 MHz might have a maximum rise time of 0.06 µs, which is 15% of the bit width.

The bit rate is also important because transmission-line effects such as ringing and reflected voltages occur during and immediately after voltage transitions, while receivers read logic levels near the middle of the bits. At slower bit rates, the bits are wider, and the voltages are likely to have settled by the time the receiver detects them.

To understand why rise and fall times are a measure of frequency, consider the most basic digital signal, a square wave, which has alternating, equal-width high and low voltages. Mathematically, a square wave is the sum of a sine wave of a fundamental frequency and its odd harmonics. For example, a 100-Hz square

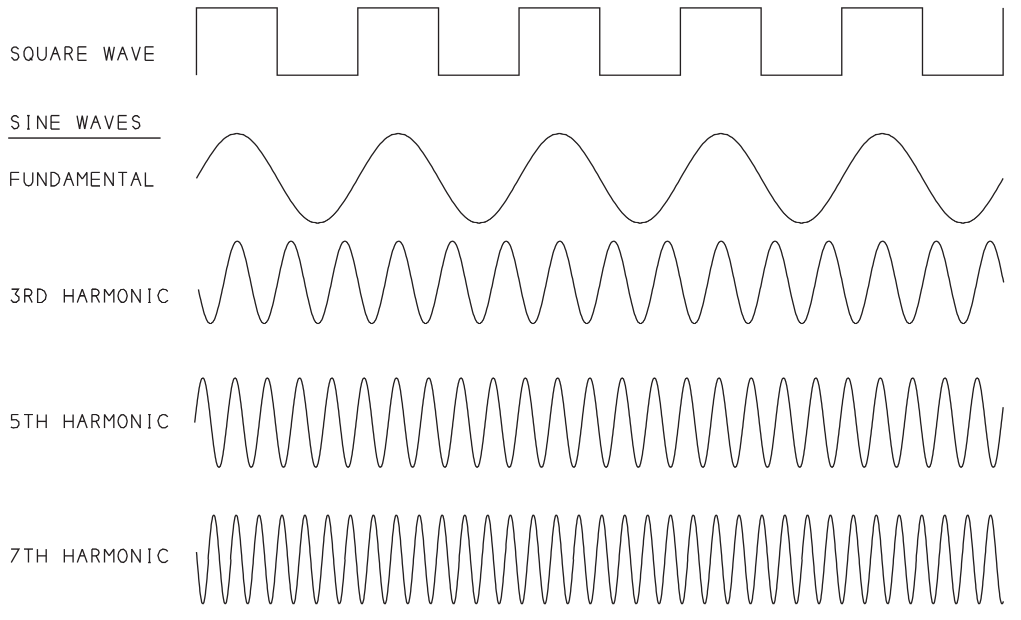


Figure 7-2: A square wave is the sum of a sine wave and its odd harmonics.

wave is the sum of a 100-Hz fundamental plus harmonics of 300, 500, 700 Hz and so on up. Figure 7-2 illustrates.

A square wave containing an infinite number of harmonics has instant transitions and rise and fall times of zero. In real life, the laws of physics limit the high frequencies and result in rise and fall times greater than zero. A signal that contains few harmonics has gradual transitions and long rise and fall times. As the number of harmonics increases, the edges sharpen and the rise and fall times shorten.

Serial data is more complex than a square wave, but the basic principle is the same: the higher the frequencies that make up the signal, the sharper the transitions.

Cable Delay

Another concept related to understanding long and short lines is cable delay. One-way delay is the time required for a signal to travel the length of the cable. The 1-way delay equals the cable's physical length divided by the propagation rate, or speed of signals in the cable.

Light in a vacuum travels 300 million meters/s (186,000 miles/s or 12 in./ns). An electrical signal in copper wire travels at around 2/3 to 3/4 this speed: between 200 million meters/s (124,000 miles/s or 8 in./ns) and 225 million meters/s (140,000 miles/s or 9 in./ns). Other terms for propagation rate are propagation velocity and transmission velocity.

These are 1-way delays for cables of different lengths:

Length (ft)	2	10	100	1000	4000
One-way delay (µs @8in./ns)	0.003	0.015	0.15	1.5	15

When the cable is short and the rise time is slow, the delays are of no consequence. But as explained below, with long cables carrying signals with fast transitions, the delays can be long enough to result in reflections that affect the logic levels at a receiver.

A cable's propagation delay, or electrical length, equals the inverse of the propagation rate and is expressed as time per unit length. The propagation delay of light in a vacuum is 85 ps/in. The propagation delay of a signal at 2/3 this speed is 125 ps/in. Another way to find the 1-way delay is to multiply propagation delay by cable length.

Calculating Line Length

The TSB-89-A supplement to TIA-485-A provides the general guideline that a line is long if the signals' rise time is less than twice the 1-way cable delay. To put it another way, the line is long if the cable's 1-way delay is greater than half the rise time. The dividing line is somewhat arbitrary. Other sources use 1/4 or 1/6 of the rise time as guidelines. RS-232 lines are always short lines because of their limited cable length and slew rate.

This function below returns true if a line behaves as a transmission line or false if the line behaves as a lumped system as determined by the propagation rate, the wires' physical length, and the driver's rise time. The variables are the propagation rate in ps/in (propagationRate), cable length in ft (cableLength), and driver rise time in ns (driverRiseTime).

VB
```
 Public Function IsLongLine _
        (ByVal propagationRate As Integer, _
        ByVal cableLength As Integer, _
        ByVal driverRiseTime As Integer) _
        As Boolean

        If (propagationRate * cableLength * 12) > _
                (driverRiseTime * 1000 / 2) Then
                IsLongLine = True
        Else
                IsLongLine = False
        End If

End Function
```

VC#
```
public bool IsLongLine(Int32 propagationRate, Int32 cableLength, Int32 driverRiseTime)
{
        bool IsLongLineReturn = false;

        if ((propagationRate * cableLength * 12) > (driverRiseTime * 1000 / 2))
        {
                IsLongLineReturn = true;
        }
        else
        {
                IsLongLineReturn = false;
        }

        return IsLongLineReturn;
}
```

The function below returns the maximum length of a short line (lumped system) for a specific propagation rate and rise time. The variables are the propagation rate in ps/in. (propagationRate), the driver rise time in ns/in. (driverRiseTime), and the maximum length of a short line in ft (MaximumLengthOfShortLine.

VB
```
Public Function MaximumLengthOfShortLine _
    (ByVal propagationRate As Integer, _
    ByVal driverRiseTime As Integer) As Integer

        MaximumLengthOfShortLine = driverRiseTime * 1000 / (propagationRate * 24)

End Function
```

VC#
```
public Int32 MaximumLengthOfShortLine(Int32 propagationRate, Int32 driverRiseTime)
{
        Int32 maximumLengthOfShortLineReturn = 0;

        maximumLengthOfShortLineReturn =
                driverRiseTime * 1000 / ( propagationRate * 24 );

        return maximumLengthOfShortLineReturn;
}
```

If the rise time is unknown, another way of deciding whether a line is long or short is to compare the shortest expected bit width and the 1-way cable delay. This method considers two factors: the reflections may bounce back and forth several times before settling, and the bit rates at the transmitter and receiver may vary slightly from each other. As a general guideline, if the bit width is 40 or more times greater than the delay, any reflections will have settled by the time the receiver reads the bits.

Choosing a Driver Chip

Two ways to reduce the effect of reflected voltages are to decrease the cable length and to increase the rise time. There's usually not much you can do about the physical length of a cable required by a particular application, but you can control the rise time with the choice of drivers.

An interface should be designed to function at the driver's minimum and maximum rise times. Table 7-1 shows examples of transceivers with different maximum bit rates and rise times. The rise time should be no more than 20% of the bit width. Remember that rise time varies with the load.

The minimum rise time determines whether or not a line is long and requires a termination. As Table 7-1 shows, with a rise time of 3 ns and a propagation rate of 125 ps/in., a cable just 1 ft long behaves like a long line.

In contrast, the MAX3080 has very slow drivers with a minimum rise time of 667 ns. The slow rise time increases the maximum length of a short line in the

Table 7-1: The maximum length of a cable that does not behave as a transmission line varies with the driver's maximum bit rate. These examples assume a propagation rate of 125 ps/in.

Chip	Maximum Bit Rate (kHz)	Rise Time (maximum) (ns)	Rise Time (minimum) (ns)	Maximum Length of Short Line (ft)
MAX3080	115	2500	667	222
MAX483	250	2000	250	83
MAX3083	500	750	200	67
MAX485	2.5	40	3	1
MAX3490	10,000	25	3	1

above example to 222 ft. The chip is rated for use at up to 115,200 bps, which is fast enough for many PC and microcontroller applications. If your application doesn't require fast bit rates, using slower drivers can help ensure signal quality.

In addition to reduced transmission-line effects, slower chips reduce the emanated EMI (electromagnetic interference) and can reduce ringing on short lines as described below.

This brings us to guideline #1 for RS-485:

Use the slowest drivers possible for the bit rate.

Line Terminations

If your cable is a long line, the proper termination will help to ensure that the receiver sees transmitted data without errors. Some short lines can also make use of terminations.

Characteristic Impedance

To properly terminate a long line, you need one more piece of information: the line's characteristic impedance. This is the input impedance of the cable if it were an infinite, open line. Every transmission line has a characteristic impedance. The value varies with the wires' diameters, their spacing in relation to other wires in the cable, and the type of insulation on the wires. The value doesn't vary with the wires' physical length but is constant for any length.

Characteristic impedance is important because a driver initially sees a transmission line as a load equal to the line's characteristic impedance. The value determines how much current flows in a line when a voltage is first applied and an output switches. When the receiver's load matches the cable's characteristic impedance, the entire transmitted signal drops across the termination and there is minimal distortion due to reflections as the voltage and current settle to final, steady-state values.

And this brings us to guideline #2 for RS-485:

Terminate long lines with the line's characteristic impedance.

Finding the Value

One way to find a cable's characteristic impedance is to obtain the value from the cable's manufacturer. Manufacturers specify characteristic impedance for products likely to be used as transmission lines. Many circuits use AWG #24 stranded, twisted pair cable, which has a characteristic impedance of 100 to 150Ω.Manufacturers can determine the value using any of several methods:

- Calculate the value mathematically from the properties of the cable. The calculation requires knowing the wires' diameter, the physical length, the distance between the wires, and the effective relative permittivity, which varies with insulation type.
- Calculate the value from measured inductance and capacitance. Using an impedance bridge, measure the line's capacitance (C) with the far end of the cable open, and measure the line's inductance (L) with the far end shorted. The characteristic impedance is $\sqrt{(L/C)}$. This calculation ignores the line's series and parallel resistance, which have little effect at high frequencies.
- Find the value empirically by applying a step function to the line and varying the termination resistor until there are no reflections. A step function is a digital pulse with a very short rise time, which ensures that the pulse contains high frequencies. Viewing the signal requires an oscilloscope with very high bandwidth. When the waveform across the termination is identical to the transmitted signal, the termination equals the characteristic impedance.

Adding a Termination

Several options are available for terminating digital lines. Table 7-2 summarizes the options.

Most RS-485 lines use an end termination consisting of a resistor equal to the characteristic impedance connected across the differential lines at or just beyond the farthest receiver. Figure 7-3 illustrates.

For a cable with a characteristic impedance of 120Ω, the termination is 120Ω across lines A and B, just beyond the A and B pins at the farthest receiver.

When two or more drivers share a pair of wires, each end of the line has a termination resistor equal to the characteristic impedance. No matter how many nodes are in the network, there should be no more than two terminations.

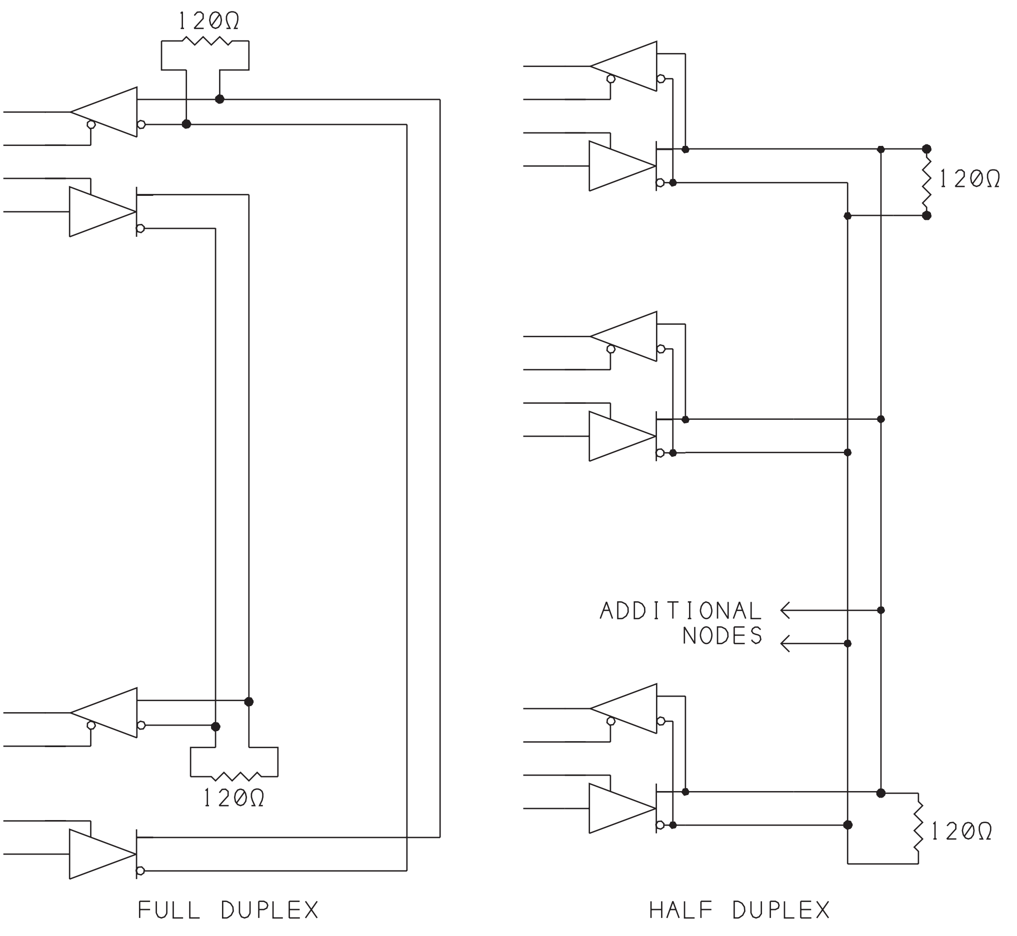


Figure 7-3: A parallel end termination requires a resistor across the differential lines at or just beyond the last receiver. A two-way interface has a resistor at each end of the line.

Table 7-2: RS-485 lines can use any of several termination options.

Termination	Advantages	Disadvantages
none	no added components, low power	suitable only for short lines with slow drivers
parallel end	effective at high bit rates	high power
series	low power	suitable only for 2-node lines
AC	low power	suitable only for low bit rates, short cables
parallel with open-circuit biasing	ensures valid logic level when open	high power, requires 2 additional resistors per bus
parallel with open and short-circuit biasing	ensures valid logic level when open or shorted	requires 4 additional resistors per node

TIA-485-A says that RS-485 drivers must be able to drive 32 unit loads plus a parallel termination of 60Ω. The total load, including the driver, receivers, and terminations, must be 54Ω or greater. On a full-duplex line, each termination resistor has its own pair of wires, so each driver sees a resistance of 120Ω. In a cable with two termination resistors across the same pair of wires, the parallel combination of two 120Ω resistors is 60Ω. The input impedance of 32 receivers at 1 unit load each decreases the total resistance of the load slightly, while the output resistance of the driver and the series resistance of the lines increase the total resistance of the load.

Effects of Terminations

You don't have to understand why transmission lines behave as they do to design a system that works. But for the curious, the following is an introduction to transmission-line theory without attempting mathematical proofs.

A transmission line has two wires, one to carry the current from the driver to receiver and another to provide a return path back to the driver. An RS-485 system is a little more complex because it has two signal wires that share a termination plus a ground return, but the basic principles are the same.

In one sense, there's nothing different about how long and short lines behave. The same laws of physics apply whether the driver is slow or fast and whether the signals travel a short or long distance. Both long and short lines can have voltage and current reflections due to an impedance mismatch.

In all cases, the reflections happen quickly, during and just after an output switches. But on a long line, the reflections are more likely to continue for enough time to cause the receiver to misread logic levels. On short lines, the reflections occur sooner and have no effect on received logic levels.

Impedance Sources of a Line

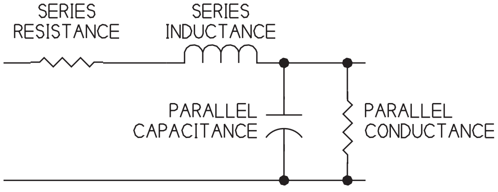
A pair of wires has several sources of impedance (Figure 7-4A). All of these sources together determine a line's characteristic impedance:

- Series resistance varies with the wire's diameter, length, and temperature.
- Series inductance varies with the wire's diameter and distance from a ground plane.
- Parallel capacitance is a measure of the electric field between the wires.
- Parallel leakage resistance is a measure of the effectiveness of the wires' insulation. The leakage resistance is typically a very high value often expressed as conductance (1/R).

One way to calculate characteristic impedance is to think of a pair of wires as a series of identical short segments with each segment having the impedance sources listed above (Figure 7-4B). To find the overall impedance of a long line, find the impedance of a short segment and use this value to calculate the impedance of an infinite series of identical segments strung together. For each added segment, the existing line is in parallel with the new segment's parallel impedance, and this combined impedance is in series with the new segment's series impedance.

As you increase the line's length, each segment added has less and less effect on the total impedance, which approaches a fixed value. This value is the imped-

(A)



(B)

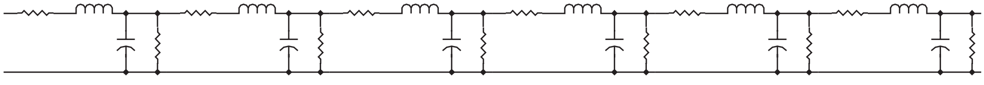


Figure 7-4: (A) A pair of wires has several impedance sources. (B) One way to find the characteristic impedance of a line is to think of the line as a series of short segments.

ance of an infinite, open line, and is equal to the line's characteristic impedance. The value is constant for any length of wire. At frequencies greater than 100 kHz, which make up most of the energy in digital pulses, the characteristic impedance is mainly resistive, so the value varies little with frequency.

Initial and Final Currents

When a voltage is first applied to a pair of wires, the voltage source doesn't know what lies at the end of the pair. The voltage source sees the load as an infinite, open line. The driver's initial current is a function of its output impedance and the line's characteristic impedance. The initial current flows even in a pair of open wires, where you might naturally assume that no current flows because the circuit is incomplete.

Shortly after reaching the end of the line, the current settles to a final value determined by the applied voltage, the termination, and series resistances in the line. If the initial and final currents vary, the line sees reflected voltages as the current settles.

Each time a driver switches state, the current transitions from an initial value to the final value.

Reflections

Figure 7-5 shows simplified examples of received voltages on lines with different terminations. In each case, what happens when the initial current reaches the end of the wires depends on what is at the end of the wires. An RS-485 driver has a low output impedance, so the impedance at the source, or driver, is less than the line's characteristic impedance.

If the termination is greater than the characteristic impedance, the signal oscillates, or rings, before settling to its final level. The same result occurs if a line has no termination except the receiver.

The extreme case of a termination greater than the characteristic impedance is when the wires are open at the far end. The open ends present a discontinuity to the current, which can't continue beyond the ends of the wires. The current has to go somewhere, so it reflects, or turns around and goes back the way it came. As the current reverses, its magnetic field collapses. The collapse increases the electrical charge and induces a voltage. The result is that the receiver sees a higher voltage than what was transmitted.

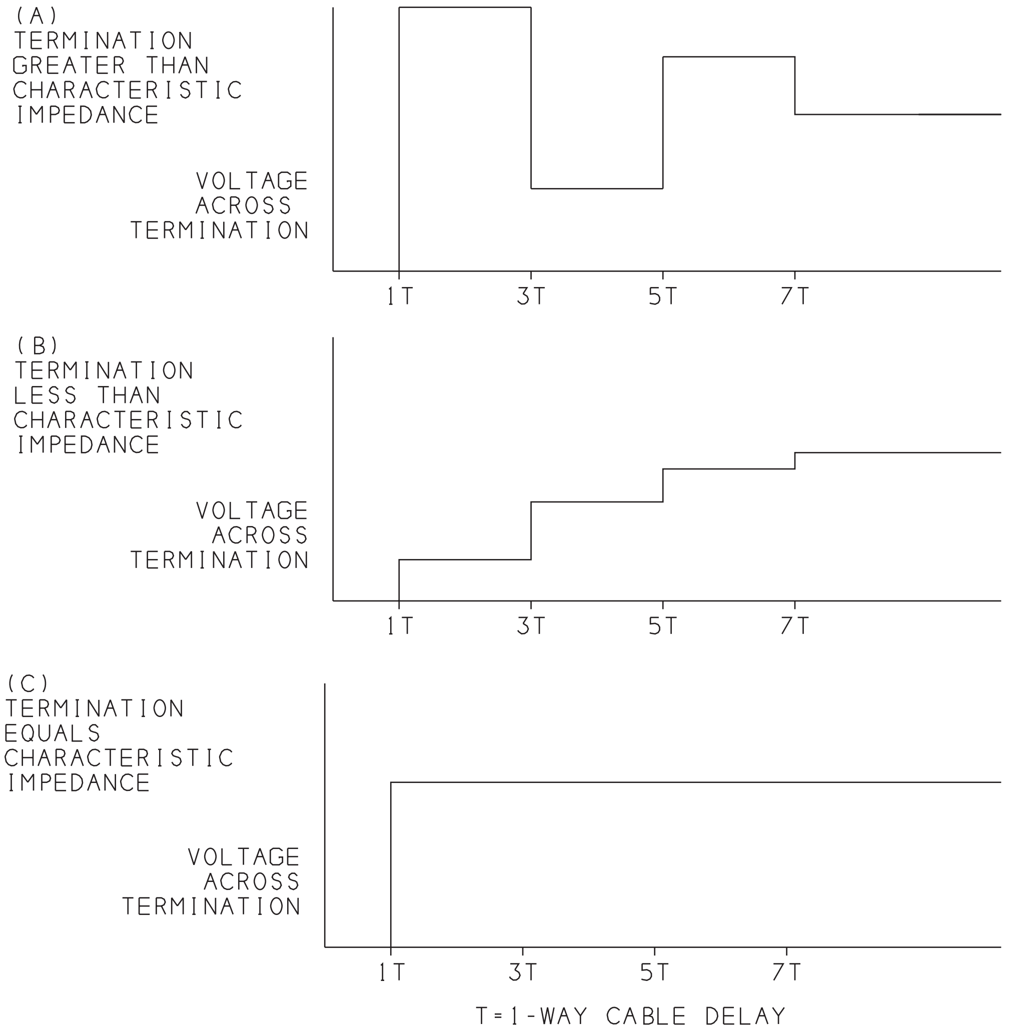


Figure 7-5: The initial voltages across a parallel termination vary depending on the difference between the termination and the line's characteristic impedance.

If the line has a termination but the value is greater than the characteristic impedance, the effect is similar but less extreme. Some of the initial current flows in the termination and the rest reflects.

The reflected current eventually returns to the driver. The driver absorbs part of the reflection and bounces the rest back, resulting in a reduced voltage at the receiver. The reflections may continue to bounce back and forth for a few

rounds with each round of lower amplitude than previous rounds. Eventually the current settles to a final value determined mainly by the termination, the driver's output resistance, and other series resistances.

If the source's impedance and termination are less than the characteristic impedance, the voltage on the line gradually rises to its final value.

The extreme case of a termination less than the characteristic impedance is when the wires are shorted together at the far end. When the current reaches the end, there is no load, so there is no voltage drop at all. The entire transmitted voltage has to reflect back to the driver. The electric field collapses and the magnetic field increases, inducing a current.

If the line has a termination but its value is less than the characteristic impedance, the effect is similar but less extreme. Some of the initial voltage drops across the termination and the rest reflects. Each time the driver re-reflects a portion of the voltage, the voltage at the receiver rises until reaching the final value.

If the wires terminate in a resistance exactly equal to the characteristic impedance, the source of the current sees no discontinuity. Instead, the source sees something that looks exactly like the infinite line the source had assumed when it applied the voltage to the line. The initial and final currents are equal, and after a single 1-way cable delay, the entire transmitted voltage drops across the resistor with no reflections at all.

Effects of Cable Length

The reflections happen fast. Increasing the physical length of a line increases the amount of time the reflections last. Each reflection bounces from the receiver to the driver and back, so each new reflected voltage arrives at the receiver after two 1-way cable delays. For example, a 10-ft cable might have a cable delay of 15 ns. A series of four reflections would last 0.12 µs. plus the initial 15 ns. Increase the cable length to 1000 ft, and the same reflections last 12 µs.

Effects of a Mismatch

If the reflected voltages are large enough and last long enough, they may have any of several effects on a line. If the receiver sees a reduced voltage, the receiver's input may drop below the threshold for the intended logic level, causing an error in the received data. If the receiver sees a greater voltage, the input transistors may saturate, slowing the response. A termination of up to 10% larger than the characteristic impedance may improve the signal quality by

increasing the initial received voltage. In extreme cases, a mismatch can cause reflections so large that they damage components.

The termination rarely matches the characteristic impedance exactly. But a value that's reasonably close reduces the amplitude of the reflections and improves signal quality overall.

Effects of a Line's Series Resistance

A line's series resistance has little effect on the characteristic impedance at high frequencies, but the series resistance can become significant for other reasons when the wires are very long. The resistance of stranded AWG #24 wire is about 25Ω/1000 ft. In a 4000-ft link, each wire has 100Ω series resistance.

If a physically long line has two 120Ω termination resistors, a large part of the signal will drop across the wires, and the receiver will see a much smaller differential voltage. However, if the signals have the minimum 1.5V difference at the driver, only a fraction of the signal needs to make it to the receiver to enable detecting the minimum required 0.2V difference. To decrease the series resistance, use wire with a lower AWG value (and thus a larger diameter).

Negative Effects of Parallel Terminations

Adding a termination is a trade off. Besides reducing reflections, terminating an RS-485 line has negative effects, including increased power consumption, lower noise margin, and overriding the receiver's internal fail-safe circuits.

The higher power consumption is a result of the line's lower series resistance. In Figure 7-6, circuits A and B show how adding a 120Ω terminating resistor decreases the parallel input impedance from 12k to 119Ω. Assuming 30Ω output impedance for each driver, the current on the line increases from 0.4 to 28 mA.

The higher current also reduces the noise margin. The driver's output impedance absorbs a larger proportion of the output voltage, reducing the differential voltage at the receivers. If the output impedance of each driver is 30Ω, one third of the voltage drops across the drivers' output impedances, leaving only 3.3V across the termination. The received differential signal is still 3.1V greater than the receiver's input threshold, however. Adding a second termination resistor exaggerates both of these effects.

One way to conserve power is to disable a driver when it isn't transmitting. If the communications path is often idle, disabling drivers that aren't transmitting

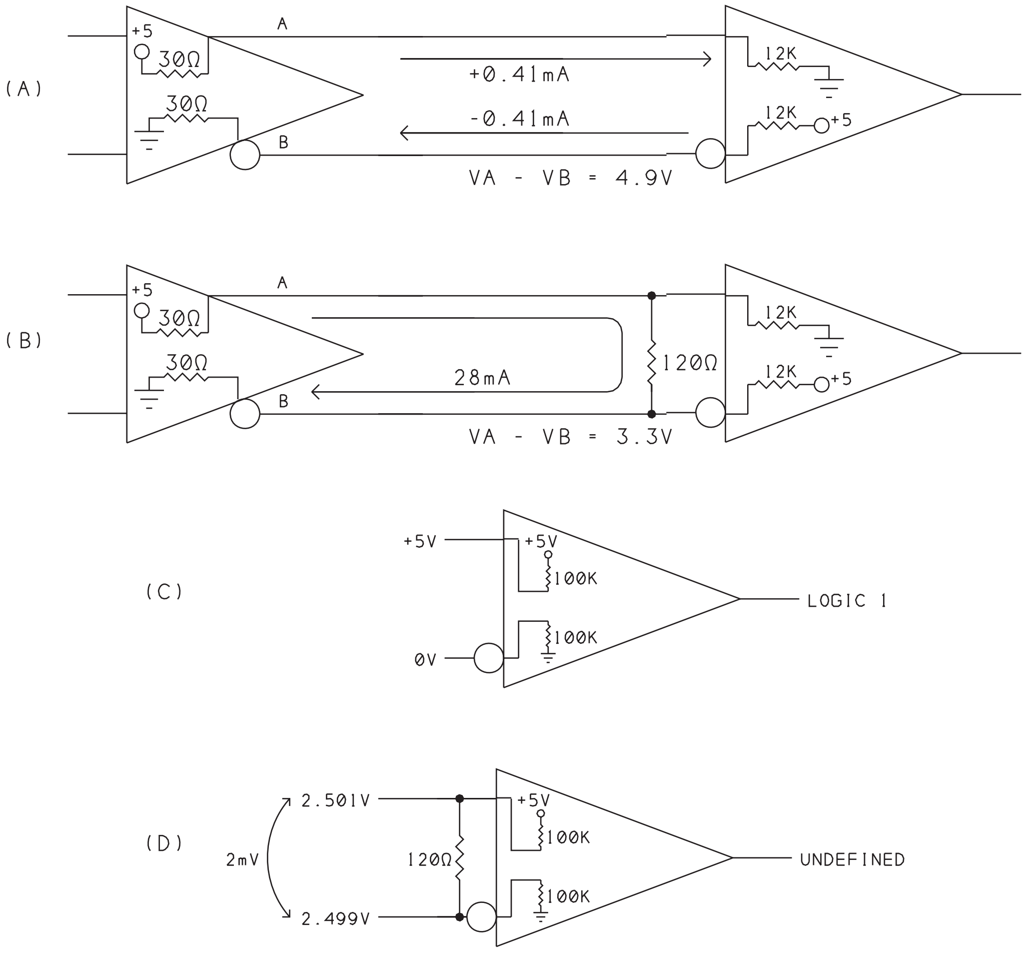


Figure 7-6: (A) and (B) show that a parallel termination increases power consumption and decreases the noise margin. (C) and (D) show how the termination defeats a receiver's internal open-circuit fail-safe circuits.

data cuts power consumption dramatically. A circuit that uses spare RS-232 signals as a power source for an RS-232-to-RS-485 converter can't use a resistive parallel termination because it draws too much current.

In Figure 7-6, circuits C and D show how adding a termination defeats the fail-safe circuits included in RS-485 receivers. The fail-safe circuits ensure that the receiver sees a defined logic level when the inputs are open. Without a termination, the internal pull up and pull down in many RS-485 receivers hold

input A more positive than input B. Adding a termination lowers the open-circuit differential voltage to just a few millivolts. This chapter shows how to add circuits that replace the internal fail-safe circuits on terminated lines.

Series Terminations

Another type of termination used in some systems is the series, or back, termination. Instead of a parallel resistor across the lines at the end of the cable, the termination resistor is at the driver, in series with the line (Figure 7-7). The termination plus the driver's output impedance equal the line's characteristic impedance.

When the output switches, half of the voltage drops across the output impedance and termination. The initial current is half as large as the final current, and the receiver sees a voltage half as large as the final voltage. The receiver's high impedance causes most of the voltage to reflect back to the driver. The driver and termination, which together equal the characteristic impedance, absorb the entire reflection. The voltage thus doubles and brings the voltage and current to their final values after just one reflection.

This type of termination can be useful in full-duplex lines between a single driver and receiver. The termination uses much less current than a parallel termination. This termination isn't recommended for networks with multiple nodes because the nodes at different locations on the line will see different reflections.

Terminations for Short Lines

If the calculations show that a line is electrically short, you might need no added termination at all. But on some short lines with fast rise times, the components form a resonant circuit that results in ringing voltages when an output switches. In these cases, a termination can ensure good signal quality at the receiver.



Figure 7-7: A series termination can absorb reflections.

The amplitude of the ringing varies with the driver's output resistance, the wires' inductance, the load's capacitance, and the frequencies carried by the wires. As with other mismatched terminations, if the ringing voltages are large enough, the receiver may misread transmitted bits.

To reduce ringing, use a driver with a slower rise time. There's no reason to use a driver capable of 10 Mbps if you're transmitting at 9600 bps. If you can't change the hardware, using a slow bit rate at least gives the ringing more time to settle before the receiver reads the input.

You can also reduce ringing by reducing the circuit's Q, which is a measurement of the circuit's ability to resonate. To reduce the Q, decrease the wires' inductance or increase the load's capacitance. To decrease the inductance, use larger diameter wires or wires that are twisted more tightly. To increase capacitance, use an AC termination as described below.

AC Terminations

An AC, or active, termination can reduce power consumption of idle lines, and may also reduce ringing voltages. However, this type of termination also reduces the maximum possible cable length and bit rate. Figure 7-8 shows examples.

In Figure 7-8A, a resistor and capacitor connect in series across the differential lines. The capacitor prevents ringing by absorbing the high frequencies that make up the ringing voltages. The capacitor also reduces power consumption because the current on the lines is near zero when the capacitor has charged after each transition. The added capacitance lowers the maximum bit rate and cable length, so this termination is limited to shorter, low-speed links.

TIA-422 gives this formula to select the capacitor's value:

CT (pF) < 2 * (1-way cable delay (ps)) / (characteristic impedance (Ω))

Assuming a propagation rate of 125 ps/in. and a characteristic impedance of 120Ω,a 10-ft cable should use a capacitor of 250 pF or less. A 100-ft cable can use values of up to 2500 pF.

In addition, the product of the terminating resistance and capacitance should be no more than 1/10 the width of a bit. For example, with 120 Ω and 2500 pF, the minimum bit width is 3 µs, for a maximum bit rate of 330 kHz.

Unlike a purely resistive parallel termination, this termination doesn't defeat the receiver's internal biasing circuits. When all drivers are off, the capacitor

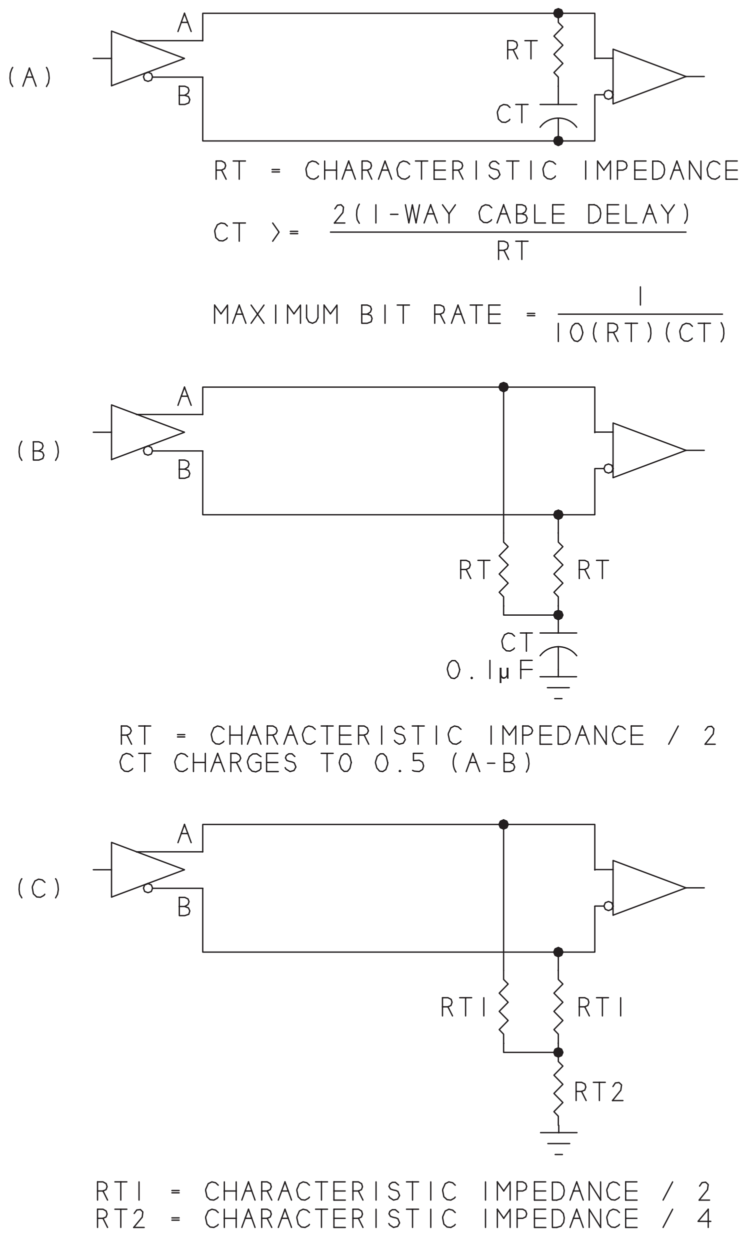


Figure 7-8: These terminations can conserve power on some lines.

remains charged and the receiver's internal pull-up and pull-down resistors hold input A > B.

In Figure 7-8B, a capacitor charges to half the differential voltage. Figure 7-8C replaces the capacitor with a resistor equal to 1/4 the line's characteristic impedance. The terminations in B and C aren't recommended when the largest source of noise is from magnetic fields.

Network Topologies

When more than two computers share a communications path, how the computers connect to the cable can also affect signal quality. Figure 7-9 shows several network topologies, or wiring configurations. RS-485 drivers and receivers are designed for use in a bus, or linear, topology, where the network cable begins at one node and connects in sequence to each of the other nodes. This topology enables using a termination at each end of the bus and brings us to guideline #3 for RS-485:

Wire the nodes in a bus topology.

A stub is the wires that connect a node to the network cable. Stubs should be as short as possible. Many sources recommend limiting stub length so the stub's 1-way delay is 1/4 to 1/2 of the signals' rise time.

Sometimes connecting the nodes along a bus isn't convenient. Wiring throughout a house lends itself to using cables that branch from one or more central locations, in a star, or hub-and-spoke, topology. An advantage to this arrangement is that if a connection opens at a node, communications among the others can continue normally.

For a setup like this, there are several options:

- Use slow drivers to increase the rise time and allow longer stubs. With the MAX3080's minimum rise time of 667 ns, a stub of 1/3 the rise time is 150 ft.
- Wire the nodes as a bus, even if this means each node has a pair of wires running to it then doubling back before going on to the next node. The line is twice as long but performance isn't compromised.
- Add a repeater circuit to regenerate the RS-485 signals where a stub connects to the main bus. The regenerated signals begin a new RS-485 line. This chapter has more about repeater circuits.

Another topology used by some networks is the ring, where each node receives from a one node only and transmits to a single, different node. Each connec-

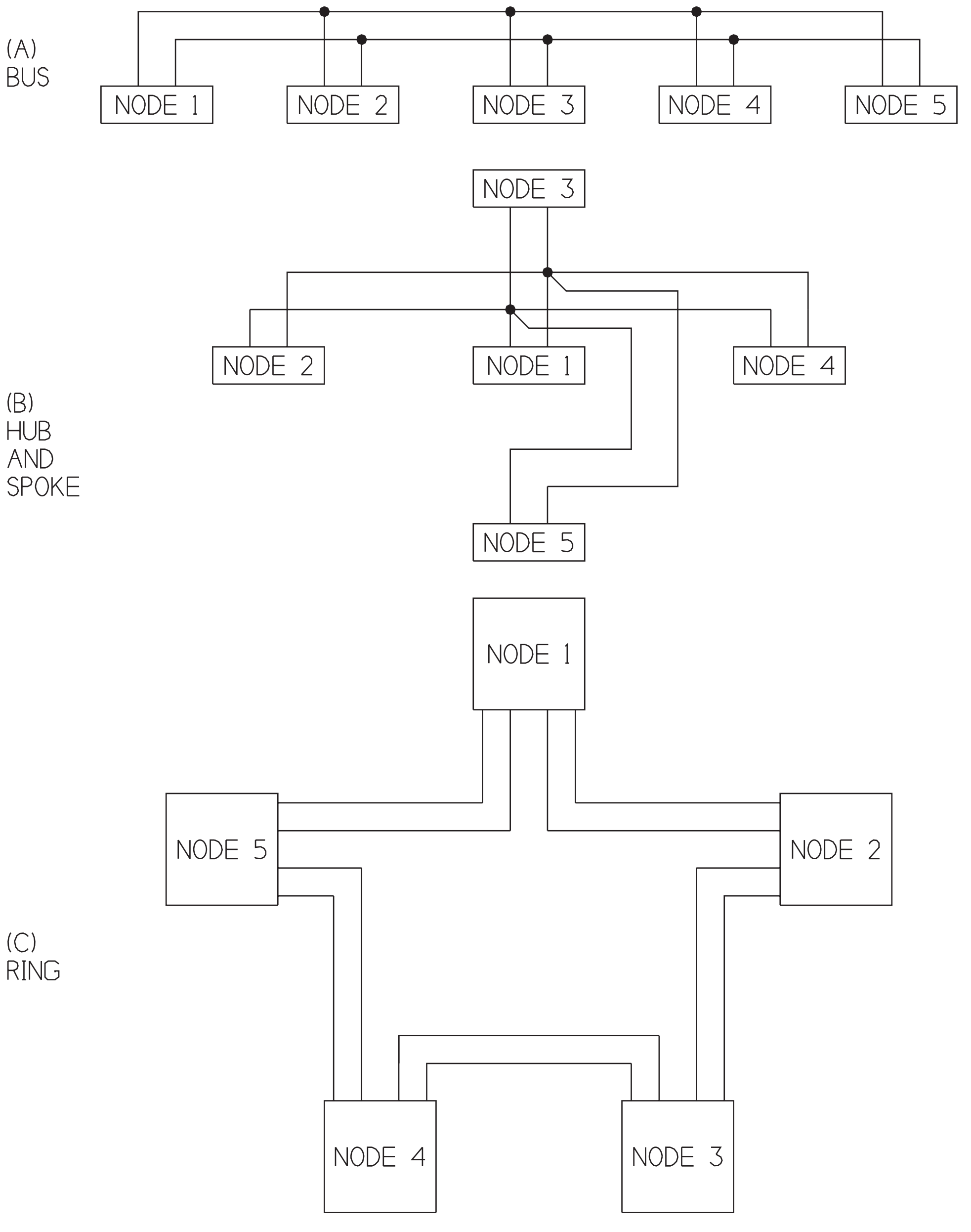


Figure 7-9: Network nodes can connect using any of several wiring topologies.

tion between two nodes is a separate RS-485 line. The distance around a ring is unlimited in theory, but the more nodes you have, the more time it takes to pass a message all the way around. An RS-485 network can use a ring topology if each node has two ports.

Biasing the Line

In a half-duplex RS-485 line, there are times when no driver is enabled. Even a full-duplex circuit might disable all of its drivers to save power when possible. In these cases, all receivers should see a logic 1, indicating an idle state. Guideline #4 addresses this need:

Bias inactive links.

Many RS-485 networks also need to ensure logic-1 inputs if the lines accidentally short together. Ways to bias links include using additional terminating components and using chips with fail-safe circuits built in.

Open-circuit Protection

In an RS-485 network, only one driver should be enabled at a time. Before enabling its driver, a node that wants to transmit must wait for any currently transmitting driver to finish transmitting. Even the busiest networks will have periods when no driver is enabled.

With no driver enabled, the signal level at a receiver's inputs might be undefined. A receiver that is enabled and detects a logic 0 will see a Start bit and will try to read a byte. The same situation exists if one or both wires accidentally open.

Most RS-485 chips include a fail-safe feature that holds input A more positive than input B when no signal is applied to the receiver. The fail-safe works fine on lines that don't use terminating resistors.

But as Figure 7-6 showed, the fail safe is defeated on lines with terminations. Figure 7-10 shows a solution that adds two 620Ω resistors: one from input A to +5V and one from input B to ground. The terminating resistor at the end with the biasing resistors is increased to 130Ω. This configuration holds terminal A about 500 mV more positive than terminal B when no drivers are enabled. The TTL outputs of the receivers are logic highs. When a driver is enabled, a logic low at the driver's input brings line B more positive than line A and brings the receivers' outputs low.

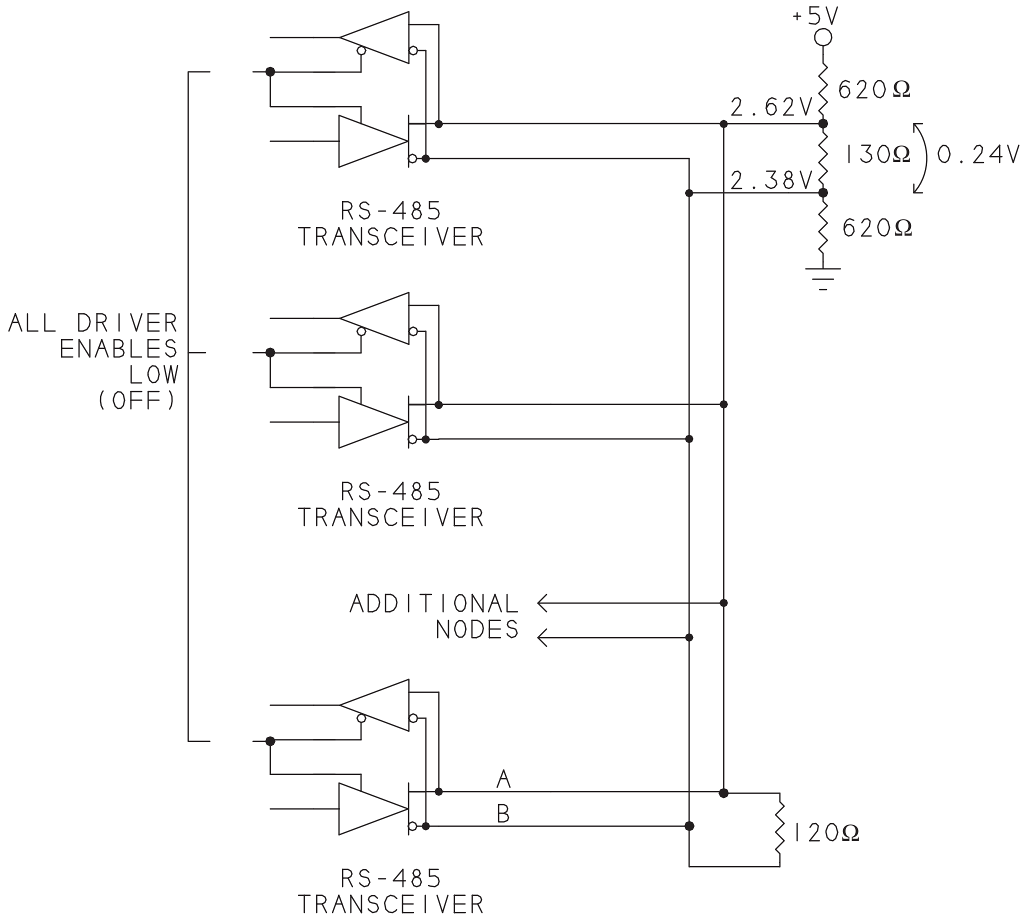


Figure 7-10: Open-circuit biasing ensures a logic 1 logic state when the line is open or no driver is active.

Biasing the entire network requires one pair of resistors. Networks that use a primary/secondary protocol typically place the biasing resistors at the primary node. If a node becomes disconnected from the network or a network wire opens, the internal fail-safe circuits hold the inputs at logic 1.

The external fail-safe components have a small effect on the current. When a driver is enabled and line A is more positive than line B, the output current is in the same direction as the bias current, so the two currents add. When the driver brings line B more positive than line A, the bias current is in the opposite direction and subtracts from the signal current. But because the drive current is

much larger than the biasing current, the opposing bias current doesn't change the logic level the receiver sees.

The biasing resistors and the parallel combination of the terminating resistors and the receivers' inputs form a voltage divider. Smaller biasing resistors increase the noise margin but also increase power consumption. With larger termination values, the biasing resistors can be a little larger. A 3V supply requires smaller values.

For an exact match, the termination at the node with the biasing resistors should be slightly greater than the line's characteristic impedance, as in Figure 7-10. At the node with the biasing resistors, the terminating resistance equals the terminating resistor in parallel with the series combination of the biasing resistors. The function below accepts a desired termination resistance and a value for the bias resistors and returns the value to use for the termination resistor at the node with the biasing resistors. All values are in ohms:

VB

```
Public Function TerminationResistor _
            (ByVal desiredTerminationResistance As Integer, _
            ByVal biasResistor As Integer) As Integer

    TerminationResistor = _
                (2 * desiredTerminationResistance * biasResistor) / _
                ((2 * biasResistor) - desiredTerminationResistance)
End Function
```

VC#

```
public Int32 TerminationResistor
            (Int32 desiredTerminationResistance, Int32 biasResistor)
{
    Int32 terminationResistorReturn = 0;

    terminationResistorReturn =
            (2 * desiredTerminationResistance * biasResistor) /
            ((2 * biasResistor) - desiredTerminationResistance);

    return terminationResistorReturn;
}
```

For example, if desiredTerminationResistance = 120 and biasResistor = 620, the function returns 133.

Biasing circuits also increase the load on the line and thus reduce the allowed number of unit loads on the bus.

Short-circuit Protection

Another concern in RS-485 links is ensuring a logic-1 input if the network wires accidentally short together or if two drivers are enabled at the same time and hold the differential voltage near 0V.

One solution is to use Maxim's MAX3080. TIA-485-A says that receivers must recognize valid logic levels when the difference between inputs A and B is at least 200mV. The MAX3080's receivers comply with the standard but expand the definition for logic 1 to include the range where input A is anywhere from 50 mV more negative to 200 mV more positive than input B. The only undefined range is when input A is between 50 mV and 200 mV more negative than input B.

With these definitions, the receiver sees a logic 1 if the difference between inputs A and B is zero, which occurs if the RS-485 wires short together. A 0V input with up to 50 mV of noise remains a logic 1.

Figure 7-11's circuit uses 75ALS180B or MAX491 driver/receivers with a resistor network to provide fail-safe protection. The chips are full-duplex driver/receiver pairs, similar to the SN75179B introduced in Chapter 6 but with an enable line for each direction. Figure 7-11's circuit is half duplex with the 2-wire interface created by tying the driver and receiver pairs together. Resistors R1 and R2 bias the line to a logic 1 if no driver is active, and R3 and

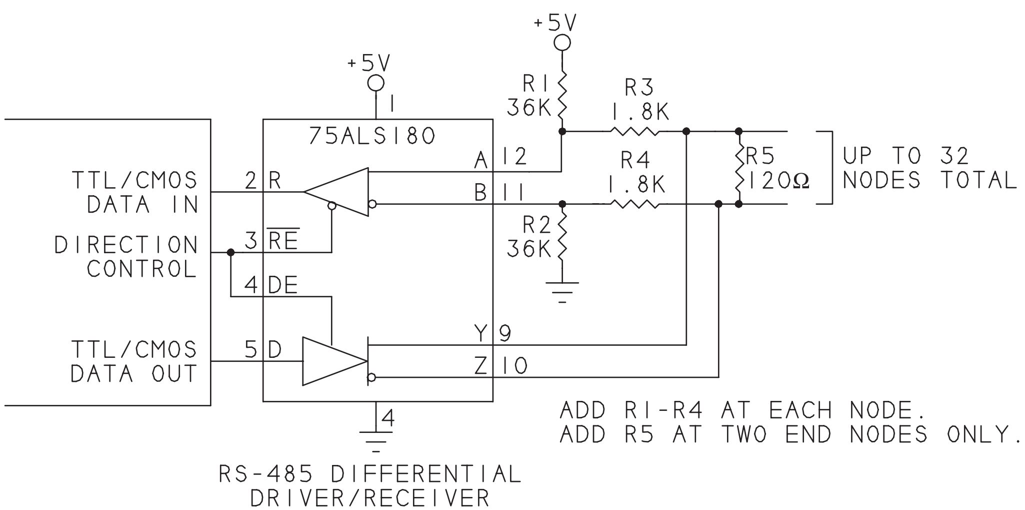


Figure 7-11: This RS-485 interface has open-circuit and fail-safe biasing.

R4 protect the receiver and ensure that the input remains biased even if the wires short together. In a network, only the two end nodes have termination resistor R5, but each node has its own set of resistors R1–R4.

This termination reduces the noise margin because R3 and R4 each drop a few tenths of a volt. If the network has fewer than 32 unit loads, you can increase the noise margin slightly by reducing the values of R1–R4. Multiply each value by half the total number of unit loads in the network. To calculate the resistor values in ohms, use:

```
R1 = NumberOfUnitLoads * 1100
R2 = R1
R3 = NumberOfUnitLoads * 55
R4 = R3
```

For example, with just two unit-load nodes, R1 and R2 would be 2.2k and R3 and R4 would be 110 Ω.

Biasing Receiver Outputs

On some half-duplex lines, including those with one control signal for the driver and receiver enable, the receiver is disabled at times, causing the receiver's TTL output to be undefined. To ensure that the output remains high, add a 10k pull-up resistor from the output to +5V. You don't need the pull up if the output connects to a microcontroller pin with an internal pull up or if the receiver drives an input to a MAX232 or other RS-232 interface chip with internal pull ups.

Cable Types

TIA-485-A doesn't recommend a specific cable type, but twisted-pair cable is inexpensive and performs well in RS-485 circuits. A twisted pair consists of two insulated conductors that spiral around each other in a double helix (Figure 7-12). The pairs typically have one or two twists per inch. Catalogs may list this type of cable as network cable or alarm wire. In a twisted pair, much of the noise that couples into the wires cancels out.

Another option is triaxial cable, which is like coaxial cable except with two conductors rather than one surrounded by a shield. Triaxial cable is expensive, however, compared to twisted pair.

Thus, here is guideline #5 for RS-485:

Use twisted-pair cable.

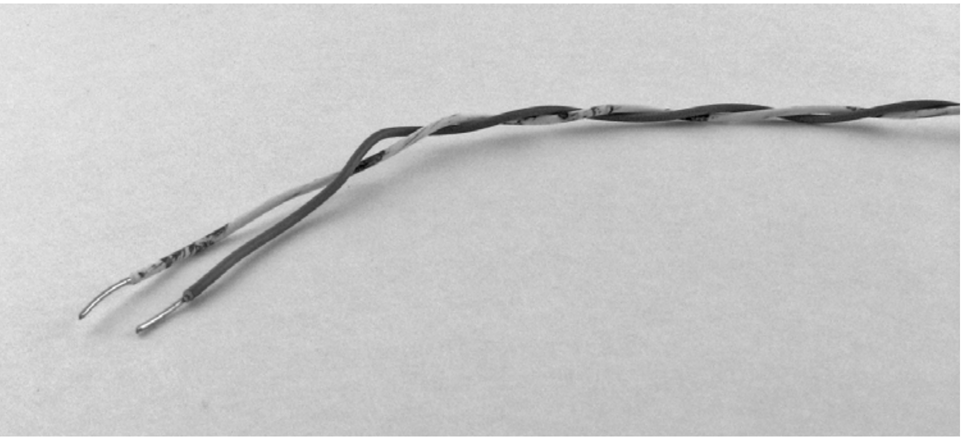


Figure 7-12: Many RS-485 systems use twisted pair cable for the data lines.

How a Wire Picks Up Noise

Understanding how a twisted pair cancels noise requires knowing something about how noise couples into a wire. Noise is any signal you don't want in a circuit. The noise can enter a wire in many ways, including by conductive, common-impedance, magnetic, capacitive, or electromagnetic coupling.

Conductive and common-impedance coupling require direct contact between the signal wire and the wire carrying the noise. Conductive coupling occurs when a wire brings noise from another source, such as a noisy power-supply line, into a circuit. Common-impedance coupling occurs when two circuits share a wire, such as a common ground return. In RS-485, the differential signals cancel much of these types of noise.

The other types of coupling result from interactions between the electric and magnetic fields that emanate from the wires themselves or that couple into the wires from outside sources.

Capacitive and inductive coupling are a source of crosstalk, where voltages in one wire couple into another. When two wires carry charges at different potentials, an electric field exists between the wires. The strength of the field varies with the distance between the wires. This electric field is the source of capacitive, or electric, coupling. Current in a wire causes the wire to emanate a magnetic field. Inductive, or magnetic, coupling occurs when magnetic fields of two wires overlap, causing the energy in one wire's field to induce a current in the other wire.

When wires are greater than 1/6 wavelength apart (1/4 mile at 10 MHz), the effects of the capacitive and inductive fields are considered together as electromagnetic coupling. An example of electromagnetic coupling is when a wire acts as a receiving antenna for radio waves.

Twisted-pair Cable

Twisted pairs are effective at canceling low-frequency noise caused by magnetic coupling. In a twisted pair, each twist of the cable swaps the physical positions of the wires. Any noise that magnetically couples into one wire is canceled in the next twist by an equal, opposite noise in the other wire.

If the twisting isn't perfectly uniform, the canceling will be less than 100%, but the noise will be much reduced. The twisting is most effective in reducing magnetic coupling of low-frequency signals, such as 60-Hz power-line noise. For a similar reason, twisted pairs also reduce the electromagnetic radiation emitted by a pair.

The magnetic field emanating from a circuit is proportional to the area between the conductors. Twisting the wires tightly reduces this area, and thus the size of the magnetic field and the amount of noise that couples into it.

With cable containing two twisted pairs, you can use one pair for the RS-485 signals and the other for a ground connection. Connect both wires in the pair to ground.

Selecting Cable

IBM and the EIA have published specifications for cable types. The specifications help in obtaining cables of known quality. Each cable type has a defined characteristic impedance and maximum bit rate. The propagation velocity may be specified as well. Manufacturers also publish specifications for cables designed for use in data links.

Many RS-485 links use 120Ω cable, but cables with higher characteristic impedance can also work. IBM Type 1 cable contains shielded twisted pairs, is rated for use at up to 16 Mbps, and has a characteristic impedance of 150Ω.

Some cable intended for data, including the popular Category 5 and Category 6 network cables, use 100Ω, unshielded twisted pairs. These cables are fine for 1-way or full-duplex RS-485 or RS-422 lines. RS-422 allows just one driver and often uses 100Ω cable with a single 100Ω termination. A 2-way, half-duplex RS-485 line would need two 100Ω resistors in parallel, which brings the parallel

combination to 50Ω, less than TIA-485-A's specified minimum. Most RS-485 drivers can source and sink 60 mA and thus will work with 100Ω cable and terminations. But a cable with 120Ω or greater characteristic impedance is a better choice for most RS-485 lines.

Shielding

Metal shielding is effective at blocking noise due to capacitive, electromagnetic, and high-frequency magnetic coupling. The shielding is typically grounded at one end only. If the line has a single power source, the shield ground is at this node. Many RS-485 lines don't require shielded cable.

Connectors

Unlike RS-232, the RS-485 standard doesn't specify a connector, signal functions, or pin assignments. Many links use RJ-type modular connectors (described in Chapter 5). On any connector, keep the two signal wires (A and B) next to each other.

The two differential lines for each signal should be in the same twisted pair. Also be careful not to transpose the wires: all of the drivers' and receivers' A pins should connect to one wire, and all of the B pins, to the other.

Grounds and Differential Lines

An RS-485 line can extend for thousands of feet. The differential drivers cause equal and opposite return currents that essentially cancel each other, and thus it may seem that RS-485 has no need for a ground connection at all. With few exceptions, however, the entire line should share a ground connection. Chapter 5 introduced the topics of power supplies, grounding, and isolation. This chapter looks at grounding and isolation as it relates to RS-485 lines, which may extend much farther than RS-232.

Ensuring a Common Ground

The currents in RS-485 balanced lines are nearly but not exactly equal. The currents differ slightly due to imbalances between the components and noise that isn't exactly equal in both wires. The current in the ground wire may be very small but isn't zero. If there is no ground connection, the energy in the return current has to dissipate somehow, possibly as radiated energy that shows up as EMI.

In some RS-485 circuits, all of the nodes and the cable that connects them share a common ground. In other circuits, the line is isolated from the nodes it connects to. In either case, all of the drivers and receivers should share a ground connection, which can have any of several sources. Most obviously, the RS-485 cable can include a wire that connects to signal ground at each node. Or the nodes' power supplies can share a common ground either through electrical wiring or via an earth ground. In a very short line, multiple nodes can share a power supply.

The specifications for RS-485 interface chips limit the permitted difference in ground potentials. Isolating the line is sometimes easier than ensuring that earth grounds at distant nodes are within the required limits.

Common-mode Voltage

TIA-485-A doesn't specify the voltage between each output and signal ground except to say that the common-mode voltage must be 7V or less. The common-mode voltage is the mean, or average, of the voltages on the two differential lines referenced to signal ground. The common-mode voltage measured at the receiver is a function of the driver offset voltage (1/2 the sum of the voltages at the A and B outputs), the difference in ground potentials at the driver and receiver, and any noise that appears equally on both lines. If the interface is perfectly balanced, the inputs are offset equally from 1/2 the supply voltage. Any imbalance raises or lowers the offset.

To comply with TIA-485-A, components must work properly with common-mode voltages from -7V to +12V. In addition, each of the receiver's inputs must be in the range of -7V to +12V referenced to the receiver's ground.

With differential signals as large as ±5V, the ground potentials at the driver and receiver can vary as much as ±7V and still comply with the standard's common-mode limit. The data sheets for interface chips specify a common-mode limit, which is often larger than the minimum requirement.

This brings us to guideline #6 for RS-485:

Limit common-mode voltages.

If the ground potentials of the driver and receiver are equal, the common-mode voltage at the receiver is the mean of the two inputs, or +2.5V with a 5V supply.

The common-mode voltages also remain within the limits when the ground potentials of two nodes vary by up to ±7V. A difference in grounds is a result of any DC differences in the ground potentials and any AC oscillations or spikes

in the ground connection. For example, if the driver's outputs are +5V and 0V relative to the driver's ground, and the driver's ground is 7V more positive than the receiver's ground, the receiver's inputs relative to the receiver's ground will be +12V and +7V (ignoring losses and noise in the differential lines). The common-mode voltage at the receiver's inputs is:

((DriverOutputA - DriverOutputB) / 2) + DriverGroundVoltage - ReceiverGroundVoltage

or

((+5 - 0) / 2) + 7 = +9.5

which is within the +12V limit.

In the other direction, if the driver's outputs are +5V and 0V and the driver's ground is 7V more negative than the receiver's ground, the receiver's inputs relative to the receiver's ground will be +2V and -7V. The common-mode voltage is:

((+5 - 0) / 2) - 7 = -4.5

which is within the -7V limit.

Why a Common-mode Voltage Limit?

Understanding the reason for the common-mode voltage limit requires looking inside the chips. Figure 7-13 shows the internal circuits for a portion of a two-way, half-duplex link. The components are as presented in National Semiconductor's application note AN-409. A wire connects the outputs of the two drivers. The receivers, termination, and the rest of the drivers' circuits aren't shown, and a complete link would include a similar circuit for the other wire in the differential pair.

A second wire connects the grounds of the two nodes. Each driver has a parasitic diode connection (D3 and D4) between the chip's grounded substrate (base material) and the collector of the output transistor. The parasitic diode is a result of the physics of the semiconductor material that makes up the chip. The chip's ground pin also connects to the substrate.

Schottky diodes D1 and D2 prevent damaging substrate currents from flowing when one of the drivers is on and the other is off. For example, if driver Y's ground potential is 5V less than driver Z's, if D1 and D2 are replaced by direct connections, current could flow in a loop through D4, Q1, and back to D4. Series resistors in the ground wire would limit the current, but driver Y's output voltage would clamp at -0.7V due to the voltage drop across D4. Diode D2

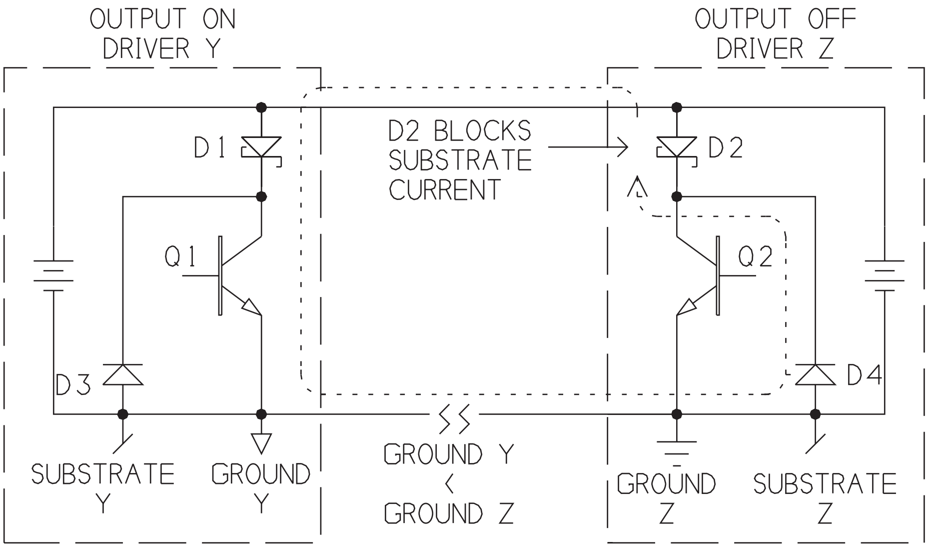


Figure 7-13: Schottky diodes in RS-485 drivers block large substrate currents between an active driver and disabled drivers.

blocks this substrate current and allows the active driver to co-exist with disabled drivers.

The protection is guaranteed only when the common-mode voltages are within the chip's specified limits. RS-422 interface chips don't have the protection diodes. For this reason, RS-422 allows only one driver per line.

Adding a Ground Wire

One way to ensure that a ground path exists between nodes is to include a ground wire in the cable (Figure 7-14). TIA-485-A recommends connecting a 1/2-Watt, 100Ω resistor in series between each node's signal ground and the ground wire. The resistors protect the components by limiting current in the ground wire if the ground voltages vary.

Isolated Lines

RS-485 cables can be much longer than RS-232 cables. Over long distances, the nodes' grounds may vary by many volts. Chapter 5 introduced galvanic isolation as a way of making a circuit immune to ground noise in other circuits.

As with RS-232, if the nodes along an RS-485 line have a common earth ground and a ground wire, ground currents from all sources will choose the

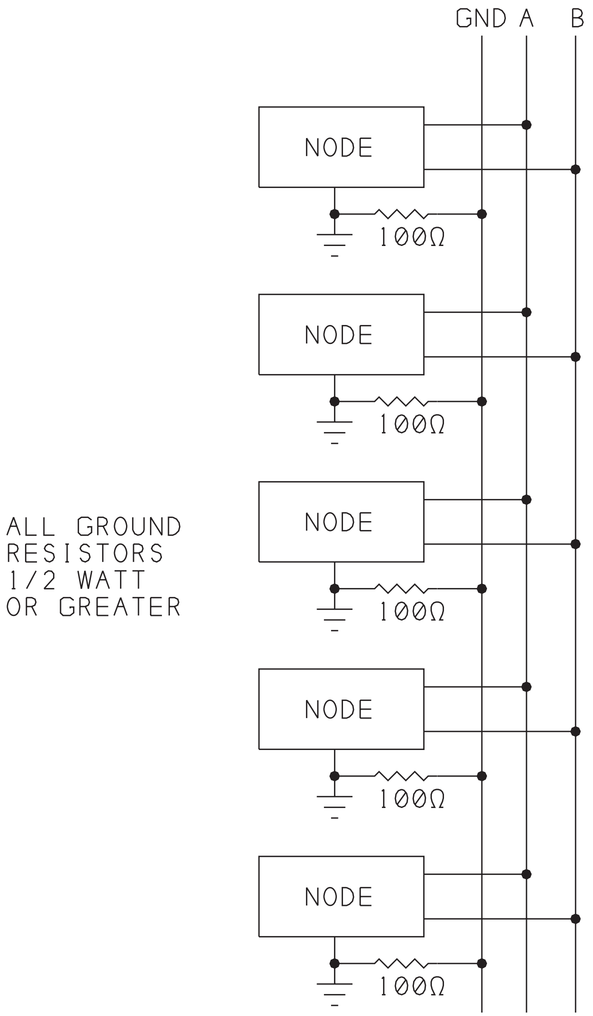


Figure 7-14: A 100Ω resistor between each node and the ground wire limits the current in the ground wire when two nodes' grounds vary.

path of least resistance. If the power supplies of all nodes use the same electrical system and their ground wires connect at an earth ground, the ground connection might be quiet. Even here, though, motors, switches, and other electrically noisy equipment can induce ground noise. If the nodes are in different buildings or using different power systems, the earth ground is likely to have higher impedance, and ground currents from other sources might find their way into

the cable's ground wire. Isolating the link can reduce or eliminate these problems.

TIA-485-A says that RS-485 lines must have a common ground. If you can't guarantee that the grounds of the nodes will be within the components' common-mode limits or if you don't want to worry about earth-ground noise, galvanic isolation is a solution.

Figure 7-15 shows four ways to isolate an RS-485 line.

Figure 7-15A has full isolation. Each node's interface has an isolated power supply and an optoisolated data line. The data lines' ground wire has no connection to any node's signal ground or earth ground. This arrangement protects the data signals from noisy earth grounds and from variations in ground voltage at different nodes. The isolation also protects the nodes from noise picked up by the line's ground wire. The nodes themselves can share an earth ground or not.

To isolate a line, you can use discrete components or a chip designed for this purpose. Maxim's MAX1480 is a complete, isolated RS-485 interface that contains a tiny transformer that isolates the line's power supply and has optical isolators for the signal lines.

Partial isolation can be cheaper or more convenient than full isolation and in some cases is sufficient.

In Figure 7-15B, the nodes and the line are isolated from earth ground, but the RS-485 line isn't isolated from the nodes it connects to. The power supplies can be batteries or floating AC supplies. This arrangement is useful if the nodes' circuits are relatively quiet but you want to isolate the nodes and line from variations in earth ground. A system where each node is battery powered has this type of isolation.

In Figure 7-15C, the data lines are isolated, but the grounds aren't isolated. This partial isolation offers some protection to the nodes if a voltage surge hits the line. Because the line shares its ground with the nodes, the grounds must be within the common-mode limits of the components. If for some reason the line can have only two wires, the line can use a common earth ground instead of a ground wire as the return path.

Figure 7-15D shows another partially isolated line. The line shares its ground with just one node, while all of the other nodes are isolated from the line. Because the line has a single ground connection, the common-mode voltage is small.

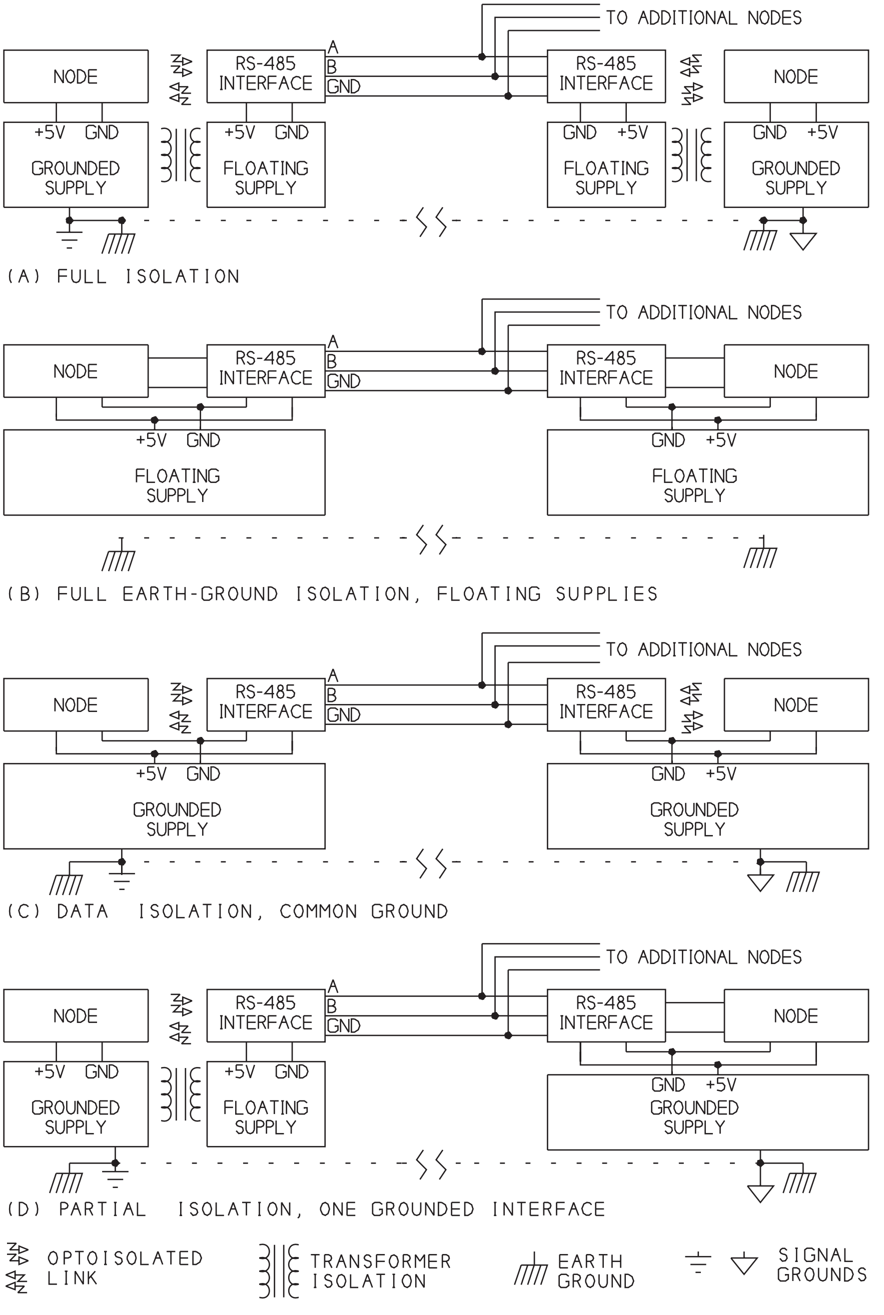


Figure 7-15: RS-485 lines can use a variety of methods to isolate data and ground.

Using Multiple Buses

A typical half-duplex RS-485 network has one pair of signal wires that connect to every node. Some networks can increase their capabilities or efficiency by using multiple RS-485 buses. Examples includes networks that have large numbers of nodes, span very long distances, or have a physical layout that benefits from a star topology.

Adding a Repeater

TIA-485-A says that a bus can have up to 32 unit loads. As Chapter 6 explained, if you need more than 32 transceivers, you can use transceivers that are less than a unit load each. Every node adds capacitance to the bus, however, and interface chips that are a fraction of a unit load often don't support fast bit rates.

Another option is to add a repeater that regenerates the RS-485 signals, allowing up to 32 more unit loads. You can also use a repeater to extend the length of a network or to add a spoke to a bus.

Figure 7-16 shows an optoisolated repeater that contains two half-duplex RS-485 transceivers. The TTL/CMOS input of each transceiver loops through a pair of optocouplers to the TTL/CMOS output of the other transceiver. RS-485 data received on one transceiver transmits out the other transceiver's RS-485 line. Each of the two RS-485 lines can support up to 32 unit loads. To eliminate having to provide a separate driver-enable signal for the repeater, both transceivers derive their driver-enable signals from the data as described in Chapter 6.

Implementing a Star Topology

Another use for multiple buses is to configure a network in a star topology. Figure 7-17 shows a network with a primary node and multiple secondary nodes. Each secondary node has its own physical bus and can branch from the primary node in any direction without having to double back to connect to other nodes. Each bus can also have multiple nodes.

Each secondary node receives everything sent by the primary node but receives nothing sent by the other secondary nodes. The resulting reduced traffic at the secondary nodes lightens their processing load.

The primary node has a full-duplex TTL/CMOS serial port, which can be a microcontroller port or a TTL/CMOS interface on an RS-232 interface chip,

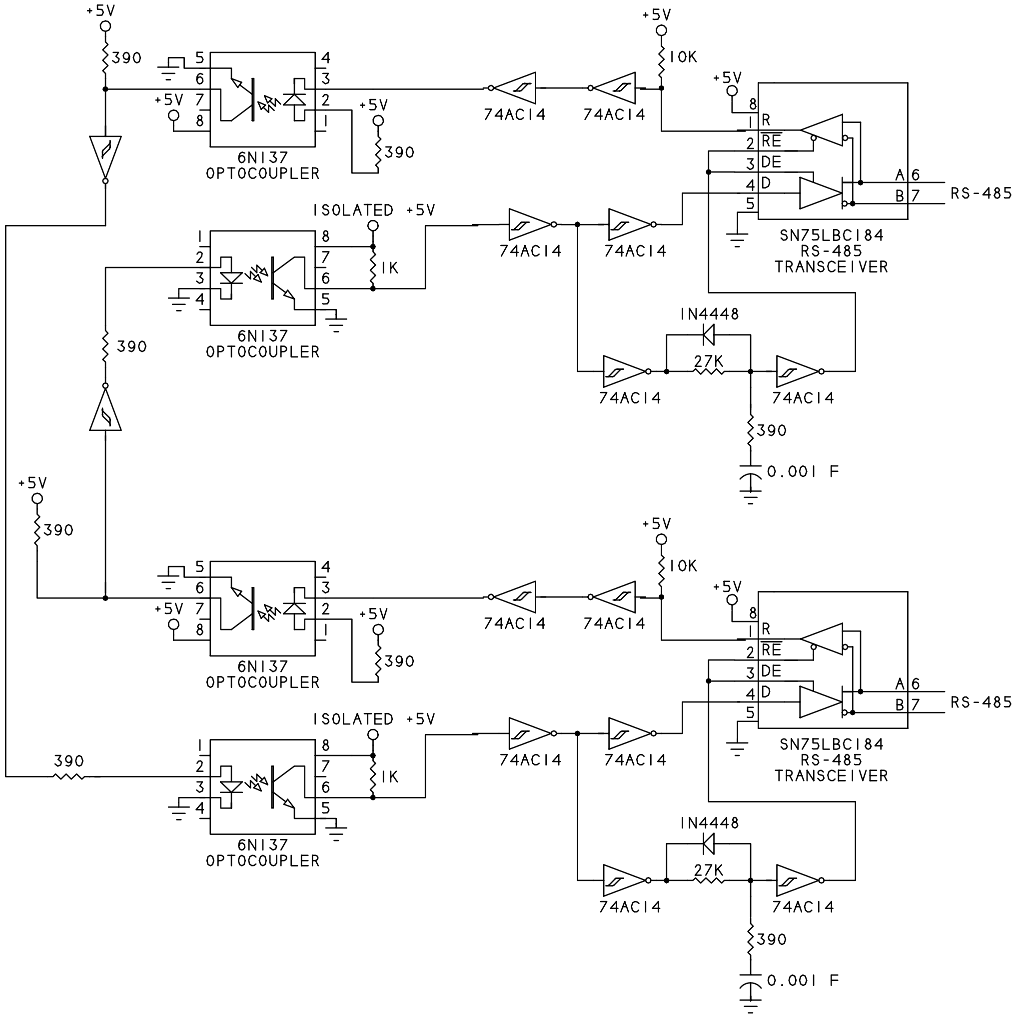


Figure 7-16: A repeater regenerates the RS-485 signals and enables the network to add nodes and cover longer distances. Circuit design courtesy of R.E. Smith (www.rs485.com).

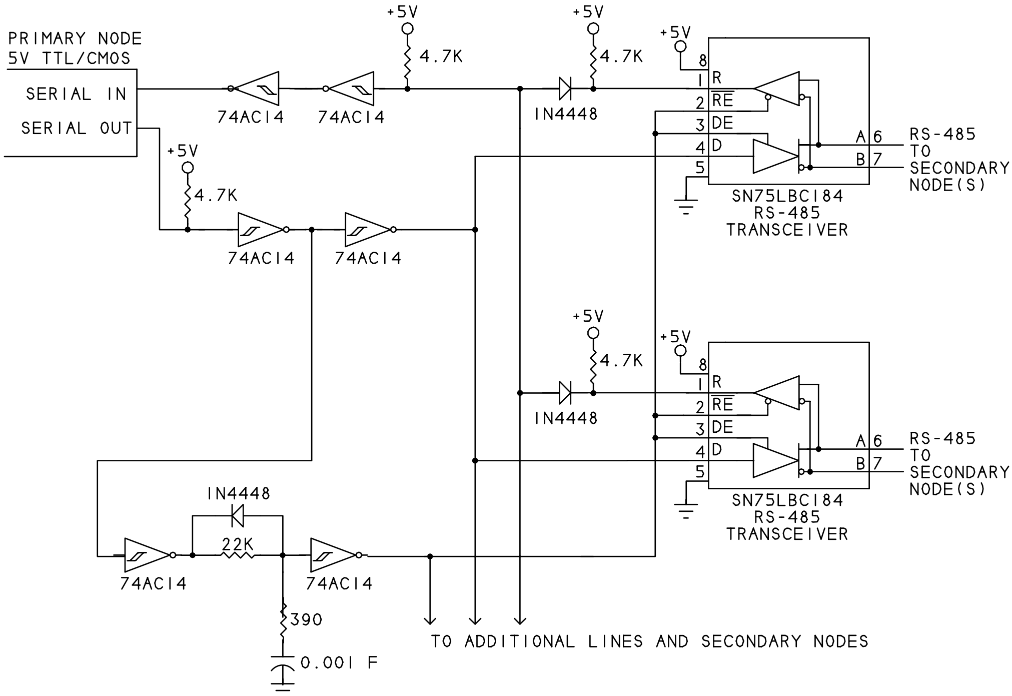


Figure 7-17: With this configuration, the primary node sees data from all of the secondary nodes and the secondary nodes see traffic from the primary node only. Circuit design courtesy of R.E. Smith (www.rs485.com).

RS-485 transceiver, or other component. The primary node's serial port connects to a series of RS-485 transceivers. Each transceiver in turn connects via RS-485 to one or more nodes in the network.

When the primary node transmits, the data passes through all of the transceivers to the secondary nodes. When a secondary node transmits, a transceiver passes the data to the primary node but not to the other secondary nodes. When all of the transceivers' R outputs are logic highs, a 4.7k resistor holds the primary node's input high. When a transceiver's R output goes low, the primary node's input goes low.

All of the transceivers derive their driver-enable signals from the data as described in Chapter 6. Using this method to control the driver-enable signals greatly simplifies the network programming for the circuit.

This page intentionally left blank

8

Going Wireless

Communicating without wires has great appeal. Wireless data can pass through walls and travel for miles. Portable devices are easier to use when there are no data cables to attach and route. Plus, wireless communicating is fun. Even in an age when wireless phones and networks are routine, there's something entertaining about designing a system that sends and receives information with no visible connection between the sender and receiver.

A data-communication system must carry information to the receiver at the desired speed, without errors, and without causing interference to other electronic devices. For many applications, wired interfaces meet these requirements easily and at low cost. Where cables aren't possible or desirable, wireless technologies are an alternative.

This chapter presents options for selecting and implementing wireless technologies for transferring asynchronous data.

Media and Modulation

Designing a wireless communication system requires deciding on a transmission medium, a modulation method if needed, and a protocol for transferring data.

Wireless data communications typically use infrared (IR) or radio-frequency (RF) energy. Many wireless systems can transmit and receive data in the asynchronous format supported by UARTs. Some wireless protocols use other data formats for greater data throughput, better reliability, or other benefits. When needed, an intelligent converter module can translate between asynchronous data and another protocol.

Using a Carrier Frequency

Instead of transmitting raw data pulses, many wireless systems transmit a carrier frequency with the data encoded as variations in the carrier's amplitude, frequency, phase, or a combination of these. The process of encoding information as variations in a carrier is called modulation. Using a carrier helps the receiver isolate transmitted data from noise and can increase the distance the data travels.

For basic data communications, two popular modulation methods are on/off key (OOK) and frequency shift key (FSK). Figure 8-1 shows both methods.

In OOK modulation, the two logic states are defined as carrier present and carrier absent. Another term for this type of modulation is carrier-present carrier-absent (CPCA) modulation. Systems that don't transmit data continuously

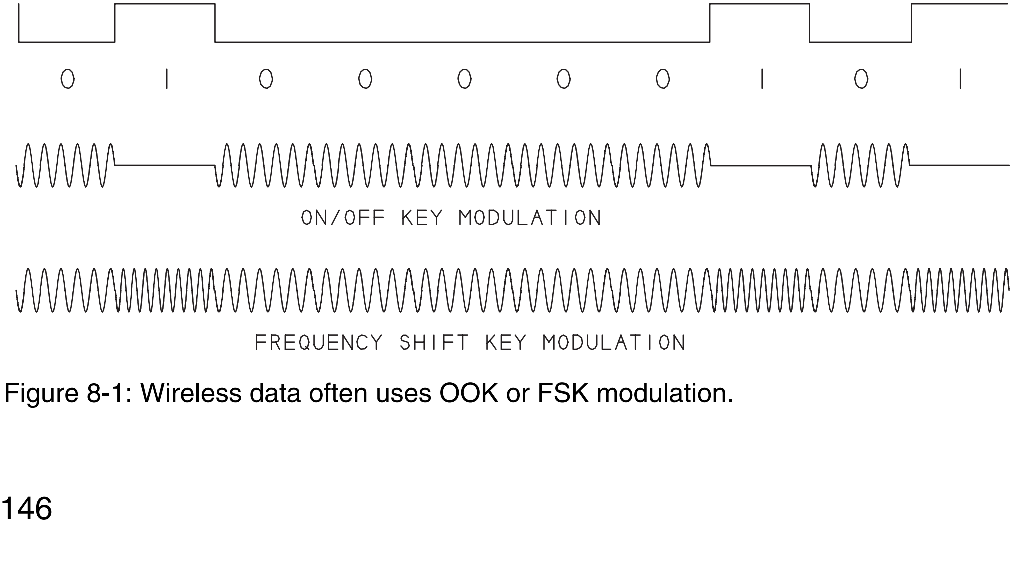


Figure 8-1: Wireless data often uses OOK or FSK modulation.

can save power by defining the data's idle state as the carrier's off (absent) state. In typical asynchronous communications, the idle state is logic 1. To transmit a logic 0, the carrier transmits for one bit period, and to transmit a logic 1, the carrier remains off for one bit period.

At the receiver, a demodulator extracts the transmitted data from the received signal. Using the example above, the receiver's output is logic 1 when no carrier is detected and logic 0 when the carrier is detected.

In FSK modulation, the carrier shifts between defined frequencies that represent logic 0 or logic 1. Early 300-bps modems used FSK modulation to transmit digital data as bursts of tones. In these modems, the originating channel uses a carrier frequency of 1170 Hz. To transmit a logic 0 bit, the carrier frequency shifts down to 1070 Hz for one bit width. To transmit a logic 1 bit, the carrier frequency shifts up to 1270 Hz for one bit width. The answering channel modulates its data in the same way but using a 2125-Hz carrier. RF systems can transmit digital data by shifting a carrier frequency in a similar way.

Spread Spectrum Technology

Some RF transmitters use spread-spectrum technology, where the transmitter uses a mathematical code or pattern to spread transmissions over a wide frequency band. The receiver uses a complementary algorithm to extract the data. Spread-spectrum transmissions have high resistance to interference and are very secure from eavesdropping but require more complex circuits than other transmission protocols.

Ensuring Reliable Transfers

Because wireless transmissions aren't confined to a cable, wireless systems need to take special care to ensure reliable transmissions. Wireless systems can implement protocols to help data reach its destination without errors and to help the receiving computer detect any errors that occur.

Managing Communications

In a typical, basic wireless system, only one transmitter sends data at a time. A primary/secondary or other protocol can help ensure that only one transmitter is active at a time.

As Chapter 2 explained, wired links often use additional lines for flow control. Because of the expense of adding wireless channels, wireless communications

typically don't have dedicated channels for sending flow-control data. Buffers or software flow control can help ensure that a transmitter doesn't send data when the receiver isn't ready.

Preventing and Detecting Errors

Radiated energy from sources other than the transmitter can appear as noise at the receiver and cause errors in received data. Chapter 2 described methods for detecting errors. These methods are also useful in wireless communications. As mentioned above, modulation and spread-spectrum technologies can also help a receiver reject noise.

Infrared

For short range, line-of-sight communications, beamed infrared energy is an option. Infrared energy is electromagnetic radiation with wavelengths in the range of 760–1000 microns, slightly longer than the wavelengths of visible red light.

Transmitters and Receivers

One or more LEDs that emit IR energy can serve as a transmitter. Lenses can direct the energy in a wide or narrow beam as needed. At the receiver, a photodiode can detect the transmitted energy.

Many IR applications use OOK modulation to reduce errors caused by stray IR energy in the environment. Common carrier frequencies are 38 kHz and 455 kHz. To send a logic 0, the carrier transmits for the width of the bit. To send a logic 1, the transmitter remains off. The receiver detects and decodes pulses at the carrier frequency and ignores any other detected IR energy.

The transmitting computer can logically AND the data signal with the output of an oscillator at the carrier frequency. For the receiver, components are available that combine a photodetector that responds to IR wavelengths, a demodulator that detects pulsed IR energy at a specific frequency, and an amplifier with a TTL-compatible output. An example is the TSOP7000 IR Receiver from Vishay Semiconductors.

Reynolds Electronics (*www.rentron.com*) is a source for components and modules for IR data links. The Fyre-Fly module contains an IR transmitter and receiver that use a 455 kHz carrier. The module has an RS-232 interface and

can transmit at up to 38,400 bps over distances of up to 80 ft. A pair of these modules can replace an RS-232 serial cable.

Some IR systems don't use a carrier and instead encode each logic-0 bit as a single pulse. To save power, the encoder can transmit pulses that are shorter than a full bit's period. For example, Microchip Technology's MSC2120 infrared encoder/decoder uses pulses that are less than 20% of the bit width.

IrDA

The Infrared Data Association (IrDA) defines a hardware interface and protocols for IR communications over distances of up to 1 meter. The limited distance makes IrDA suitable mainly for communicating with nearby peripherals. The IrDA specifications define several physical layers that each specify a format for transmitted data. The Serial IrDA (SIR) physical layer supports transmitting asynchronous data at up to 115,200 bps.

Radio Frequency

For longer distances or when a light of sight isn't available or practical, radio-frequency (RF) communications are an option.

Complying with Regulations

RF communications must comply with government regulations. In the U.S., the Federal Communications Commission (FCC) administers regulations that divide the RF spectrum into bands and specify uses for each band. The regulations are in the Code of Federal Regulations, Title 47 (*http://wireless.fcc.gov/rules.html*). Part 15 of the regulations covers unlicensed transmissions.

Many small-scale applications can use bands that allow unlicensed transmissions, which don't require a licensed operator. Transmitters that use these bands must comply with regulations that apply to the bands, however. The FCC specifies the allowed radiated power for the fundamental frequency and for radiation outside the necessary bandwidth. Radiated power is specified as field strength, which is a measure of the voltage of the radiated signal at a defined distance from the transmitting antenna. For some bands, the regulations also specify the allowed type of content and purpose of the transmissions.

Commercial products that intentionally radiate RF energy must obtain certification by passing FCC compliance tests. The FCC authorizes private test labs

to perform the tests. On passing compliance tests, the FCC issues an ID number to the product.

Choosing an RF Band

Designs that don't require licensed operators can use any of several RF bands. In general, to transmit a particular distance, a lower frequency requires less power but a longer antenna. The maximum range varies with the design of the transmitter and receiver and the environment.

The transmitter modulates an RF carrier frequency with the data. The receiver demodulates the carrier to extract the data. The bands described below are suitable for transmitting asynchronous serial data.

Frequencies for Unrestricted Use

Two popular bands for unlicensed communications are 902–928 MHz and 2400–2483.5 MHz (called the 2.4 GHz band for short). FCC regulations specify the maximum transmitted field strength, harmonic levels, and spurious radiation for the bands.

Because these bands are available for just about any use, interference from other devices is more likely in these bands compared to others. Wireless networks (802.11b and 802.11g), Zigbee and Bluetooth systems, and many cordless phones use the bands. Part 18 of the federal regulations also allocates these bands for use by industrial, scientific, and medical devices.

Frequencies with Restricted Uses

The band 260–470 MHz is available for unlicensed use with some restrictions on the content, duration, and scheduling of transmissions. Devices that can operate within the restrictions have access to a generally less crowded portion of the RF spectrum.

Section 15.231 of the FCC regulations specifies the allowed types of transmissions. Subsections *a–d* allow these uses:

- Control or command signals such as alarm data, door openers, and remote switches.
- ID codes that identify a system component.
- Variable data accompanied by a control or ID code.

The duration of transmissions has limits that vary with the activation method and purpose:

- A transmitter with automatic activation must cease transmitting within 5 seconds of activation.
- A transmitter with manual activation must cease transmitting within 5 seconds of being released after use.
- Polling to determine system integrity in security or safety applications is allowed for a total of up to 2 s/hr per transmitter.
- Operation of any duration is allowed during an emergency.

Subsection *e* allows these additional uses with further restrictions:

- Voice and video.
- Toy control.
- Variable data without a control code.
- Periodic transmissions at regular, predetermined intervals (other than polling for system integrity as described in subsection *a*).

Transmissions under subsection *e* have these restrictions:

- The maximum allowed field strength of the fundamental frequency and spurious emissions are up to 60% less compared to transmitters under subsections *a–d*.
- The maximum transmission time is 1 second.
- The minimum period between transmissions is at least 10 seconds and must be at least 30 times longer than the transmission period. For example, after transmitting for 0.3 second, a device must wait 10 seconds before transmitting again. After transmitting for 1 second, a device must wait 30 seconds before transmitting again.

Implementing a Link

A quick way to implement an RF link for asynchronous serial data is to use transceivers with built-in asynchronous serial interfaces. Sources include Linx Technologies, Inc. (*www.linxtechnologies.com*) and MaxStream, Inc. (*www.maxstream.net*). Chips, modules, and complete units are available for incorporating into designs.

The Linx TXM-433-LR transmitter module has an asynchronous serial input that can connects to a UART's data output and a 433-MHz output that can

connect to an antenna. The companion RXM-433-LR receiver module has a 433-MHz receiver that can connect to an antenna and an asynchronous serial output that can connect to a UART's data input.

The modules use OOK modulation. They don't implement any error-detecting protocols so the computers that connect to the asynchronous serial interfaces are responsible for detecting errors. A link can operate over distances of up to 3,000 ft at up to 10,000 bps. Parallax, Inc. offers 433 MHz RF modules that each contain an antenna and a Linx transmitter or receiver. A header provides connections for +5V, ground, data, and a power-down input, plus a signal-strength indicator on the receiver.

MaxStream's 1.9XTend OEM RF module has built-in error detecting and the ability to transmit over longer distances. The module can interface directly to a UART. The range is up to 40 miles using high-gain antennas and outdoor, line-of-sight transmissions at 9600 bps or less. The transmitter uses spread-spectrum technology and encryption.

The modules can transmit and receive raw data or implement either of two built-in methods of error detecting with retries. A command can configure a module to look for an acknowledgement after transmitting a packet of data. If the acknowledgement doesn't arrive, the module automatically retries the transmission up to a specified number of times. Another command can configure the module to repeat all transmitted data a specified number of times.

These modules are just a sampling of what's available. The vendors mentioned above and others offer products to suit just about any application's needs.

Using Other RF Standards

Several standards define interfaces and protocols for transmitting RF data in formats other than asynchronous serial data. Modules are available that contain intelligent controllers that convert between asynchronous serial data and these interfaces. Table 8-1 compares three options.

Zigbee

The Zigbee standard (*zigbee.org*) defines a hardware interface and software protocols suitable for monitoring and control functions and other applications that transmit data in networks at up to 250 kbps. The components are inexpensive and consume little power. The protocol supports encryption.

Table 8-1: Wireless systems can use standard protocols to carry data.

Interface	Data Throughput (bps)	Maximum Distance (meters)
Bluetooth	3M	100
Zigbee	250k	50, extendable with routers
Wi-Fi	54M	40 or more

The physical network is based on the IEEE 802.15.4 standard for wireless personal area networks. The transmitters use spread-spectrum technology. In the U.S., Zigbee systems transmit in the 2.4 GHz or 915 MHz bands. A transmission can travel up to 50 meters, and routers can extend the distance.

Zigbee supports multiple network topologies, including a mesh topology where each node can communicate with any other node within range. In a Zigbee network, one device is the Coordinator, which manages the network. The other devices are either End Devices, which perform monitoring or control functions, or Routers, which extend the range of the network and can perform monitoring or control functions as well.

MaxStream's XBee-PRO modem has an asynchronous serial interface and a ZigBee interface. Data received on the asynchronous serial port transmits on the ZigBee interface, and data received on the ZigBee interface is written to the asynchronous serial port. A vendor-specific command mode and API enable configuring the modem over the asynchronous interface.

For more about Zigbee programming, I recommend the book *Hands-On ZigBee* by Fred Eady.

Bluetooth

Bluetooth (*bluetooth.org*) is a hardware and software standard suitable for a variety of wireless applications. The interface supports secure transmissions at low power and low cost. Devices can transmit data over distances of up to 100 meters and at speeds of up to 3 Mbps. Bluetooth uses the 2.4-GHz band with spread-spectrum technology. Up to eight Bluetooth devices can communicate with each other in a collection called a piconet.

Bluetooth Profiles define higher-level protocols for specific functions. The Serial Port Profile supports COM-port emulation.

A7 Engineering, Inc. (*www.a7eng.com*) is a source for Bluetooth modules with asynchronous serial interfaces. The eb501-SER module can interface directly to an RS-232, 5V, or 3V asynchronous serial port. The device firmware supports

two modes. In one mode, the modules configure themselves automatically on powering up. The computers that interface to the modules can communicate as if they had a wired connection with no software changes required. In the other mode, the computers that interface to the modules can use plain-text commands to configure and control the interfaces.

Wi-Fi

Another option for asynchronous serial data is to use a serial server with a Wi-Fi (IEEE 802.11b or other wireless-network) interface as described in Chapter 3.

9

Using .NET's SerialPort Class

Programmers who use Microsoft's .NET Framework can use the SerialPort class to access COM-port devices. Applications can use the class's properties, methods, and events to access ports without having to resort to low-level programming or the Windows API. The SerialPort class was added in version 2.0 of .NET.

This chapter shows how to use the class to exchange text and binary data. The code in this chapter uses resources in the following namespaces:

VB

```
Imports System
Imports System.IO.Ports
```

VC#

```
using System.IO;
using System.IO.Ports;
```

Gaining Access to a Port

To access a COM port, an application creates a SerialPort object, sets the communication parameters, and opens a connection to the port.

Finding Ports

The SerialPort class's GetPortNames method returns an array of the names of all of the system's COM ports. The array's elements aren't guaranteed to be in alphabetical order, but the Array.Sort method can sort if needed.

VB
```
Dim nameArray() As String

nameArray = SerialPort.GetPortNames
Array.Sort(nameArray)
```

VC#
```
string[] nameArray = null;

nameArray = SerialPort.GetPortNames();
Array.Sort( nameArray );
```

Because users can attach and remove virtual COM ports while an application is running, an application may need to look for ports more than once. Also be aware that the array index isn't the same thing as a port's COM-port number. For example, if a PC's only port is COM4, the port's SerialPort object will be at index zero in the array.

Opening a Port

Before accessing a port or setting port parameters, the application must create a SerialPort object:

VB
```
Friend myComPort as New SerialPort
```

VC#
```
internal SerialPort myComPort = new SerialPort();
```

The SerialPort object has properties for setting port parameters. The default parameters are 9600 bps, no parity, one Stop bit, and no flow control.

The object's PortName property should match a name in the array returned by the GetPortNames method described above. An application that wants to use a specific port can search for a match in the array:

VB
```
Dim index As Integer = -1
Dim nameArray() As String
Dim myComPortName As String

' Specify the port to look for.

myComPortName = "COM5"

' Get an array of names of installed ports.

nameArray = SerialPort.GetPortNames

Do
    ' Look in the array for the desired port name.

    index += 1

Loop Until ((nameArray(index) = myComPortName) Or _
        (index = nameArray.GetUpperBound(0)))

' If the desired port isn't found, select the first port in the array.

If (index = nameArray.GetUpperBound(0)) Then
    myComPortName = nameArray(0)
End If
```

VC#
```
int index = -1;
string[] nameArray = null;
string myComPortName = null;

//  Specify the port to look for.

myComPortName = "COM5";

//  Get an array of names of installed ports.

nameArray = SerialPort.GetPortNames();

do
{
        //  Look in the array for the desired port name.

        index += 1;
}
while ( !((nameArray[index] == myComPortName) |
        (index == nameArray.GetUpperBound(0))));

//  If the desired port isn't found, select the first port in the array.

if (  index == nameArray.GetUpperBound( 0 ) )
{
        myComPort Name= nameArray[ 0 ];
}
```

To change a parameter, set the property's value:

VB
```
myComPort.PortName = "COM6"
myComPort.BaudRate = 115200
myComPort.Parity = Parity.Even
myComPort.DataBits = 7
myComPort.StopBits = StopBits.Two
myComPort.Handshake = RequestToSend
```

VC#
```
myComPort.PortName = "COM6";
myComPort.BaudRate = 115200;
myComPort.Parity = Parity.Even;
myComPort.DataBits = 7;
myComPort.StopBits = StopBits.Two;
myComPort.Handshake = RequestToSend;
```

You can also set some of the parameters when initializing a SerialPort object:

VB
```
myComPort =  new SerialPort("COM6", 115200, Parity.None, 8, StopBits.One)
```

VC#
```
myComPort =  new SerialPort("COM6", 115200, Parity.None, 8, StopBits.One);
```

Before communicating with a port, the application must open a connection to the SerialPort object. The Open method uses the specified parameters or default parameters if values haven't been specified:

VB
```
myComPort.Open()
```

VC#
```
myComPort.Open();
```

VB In Visual Basic, an alternate way to open a connection uses the My.Computer.Ports object. Note that the object uses the OpenSerialPort (not Open) method:

```
myComPort = My.Computer.Ports.OpenSerialPort( )
```

The OpenSerialPort method can also accept parameters:

```
myComPort = _
    My.Computer.Ports.OpenSerialPort("COM6", 9600, Parity.None, 8, StopBits.One)
```

Other .NET languages don't support the My object and thus can't open a COM port in this way.

Opening a port tests that the port is available and that the parameters are valid. After opening a port, the application has exclusive use of the port. No other application or resource can open the same port until the port is closed. An application can change port parameters such as bit rate and parity while the port is open.

Attempting to open a port that is in use raises an UnauthorizedAccessException. To prevent the exception, application code can read the port's IsOpen property to verify that the port is closed before attempting to open it:

VB
```
If (Not myComPort.IsOpen) Then

    myComPort.Open()

End If
```

VC#
```
if (!( myComPort.IsOpen ))
{
        myComPort.Open();
}
```

Of course it's possible that another resource might open the port after reading IsOpen but before calling the Open method. In that case, the application can catch the exception and display a message or take other action as needed:

VB
```
Try
        myComPort.Open()

Catch ex As UnauthorizedAccessException

        MessageBox.Show(ex.Message)

End Try
```

VC#
```
try
{
        myComPort.Open();
}
catch ( UnauthorizedAccessException ex )
{
        MessageBox.Show(ex.Message);
}
```

The examples that follow in this and following chapters assume the application has an open port called myComPort.

Timeouts

An application might want a way to bail out if the remote port isn't ready to send or receive data. The ReadTimeout and WriteTimeout properties set the amount of time to wait in milliseconds:

VB
```
myComPort.ReadTimeout = 3000
myComPort.WriteTimeout = 5000
```

VC#
```
myComPort.ReadTimeout = 3000;
myComPort.WriteTimeout = 5000;
```

The default is the constant SerialPort.InfiniteTimeout, which never times out. Microsoft's documentation says the values must be either greater than zero or SerialPort.InfiniteTimeout. A timeout raises a TimeoutException.

The best values to use for timeouts depend on the bit rate and the size of the largest transfers expected in each read or write operation. For example, at 300 bps and parameters of 8-N-1, a 1024-byte array requires over 34 seconds to transmit. At 115,200 bps, the same array can transmit in under 0.1 second. Delays due to flow control can slow a transfer further. Chapter 10 has more about setting timeouts.

Receive Threshold

A DataReceived event can inform an application that data is available for reading in the receive buffer. The ReceivedBytesThreshold property determines how many bytes must be present in the buffer before the event is raised. The default is one byte. If you set the property to a larger number, the DataReceived event is likely to be raised less often and each event will likely have more data to process.

Setting a threshold greater than 1 can prevent the receiving computer from seeing some or all of the received data. For example, assume that ReceivedBytesThreshold in the receiving computer is set to 4 and the transmitting computer sends 3 bytes. The buffer holds fewer bytes than the threshold, so the DataReceived event isn't raised. Unless the transmitting computer sends another byte or the receiving computer polls the receive buffer, the application doesn't know the bytes arrived.

Closing a Port

An application that has finished communicating with a port should close the port and free its resources to allow other applications to use the port if needed. The Close method closes the connection to the port and clears the receive and transmit buffers. The Dispose method calls the Close method and releases all resources used by the component, making it possible to reopen a port quickly after closing it.

Attempting to close a port that doesn't exist causes a NullReferenceException. Attempting to close a port that isn't open causes an InvalidOperationException. To avoid these exceptions, check to see that the port exists and is open before closing it.

When a port is transmitting, closing the port or clearing the transmit buffer can crash Windows. To prevent crashes, wait for all data to transmit before closing the port. To prevent an endless wait, set the port's WriteTimeout property. On a timeout, the port's BytesToWrite property is set to zero, and the port can close without a crash.

VB

```
If (Not (myComPort Is Nothing)) Then

        ' The COM port exists.

        If myComPort.IsOpen Then

                ' Wait for the transmit buffer to empty.

                Do While (myComPort.BytesToWrite > 0)
                Loop

                ' The COM port is open; close it and dispose of its resourced.

                myComPort.Dispose()
        End If
End If
```

VC#

```
if (!(myComPort == null))
{
        //  The COM port exists.

        if ( myComPort.IsOpen )
        {
                //  Wait for the transmit buffer to empty.

                while (myComPort.BytesToWrite > 0)
                {
                }
                //  The COM port is open; close it and dispose of its resources.

                myComPort.Dispose();
        }
}
```

An alternate way to close a port is with a Using block. On exiting the Using block, the SerialPort object named in the Using statement is closed automatically and its resources are disposed of:

VB
```
Using myComPort As New SerialPort
        myComPort.PortName = "COM6"
        myComPort.BaudRate = 115200
        myComPort.Open()
        myComPort.WriteLine("hello")
End Using
```

VC#
```
using (SerialPort myComPort = new SerialPort())
{
        myComPort.PortName = "COM6";
        myComPort.BaudRate = 115200;
        myComPort.Open();
        myComPort.WriteLine("hello");
}
```

Closing a port can take up to a few seconds (or longer if waiting for the port to finish transmitting), so applications and users should delay a bit between closing a port and re-opening the same port.

Transferring Data

The .NET Framework supports many methods for reading and writing to COM ports. An application that sends a text command and waits for a response will likely use a different method than an application that is sending a large file of object code.

The SerialPort class provides methods for reading and writing to ports. Applications can also use a SerialPort object's BaseStream property to obtain BinaryReader, BinaryWriter, StreamReader, and StreamWriter objects and use their methods to access a port. Table 9-1 and Table 9-2 summarize methods for reading and writing to SerialPort objects.

A basic data type for COM-port data is the byte array. An application can transfer any kind of data in byte arrays. The COM-port software makes no assumptions about the meaning of the data or how the receiving computer will use the data. An application can encode data before placing it in a byte array and writing the array to the port. The receiving computer can decode the received data as needed.

Table 9-1: Applications can use a variety of methods to write data to a SerialPort object.

Object	Method	Data Type	Blocking
SerialPort	Write	Byte array. Char array or subarray. String.	yes
	WriteLine	String + NewLine. NewLine only.	yes
	WriteByte	Byte.	yes
BinaryWriter	Write	Byte, Byte array or subarray, SByte. Char, Char array or subarray. String. Boolean, Decimal, Double, Int16, Int32, Int64, Single, UInt16, UInt32, UInt64.	yes
StreamWriter	Write	Char, Char array or subarray. String, formatted String. Text representation of Boolean, Decimal, Double, Int32, Int64, Object, Single, UInt32, UInt64.	yes
	WriteLine	Same as Write but with NewLine appended. NewLine only.	yes

For applications that transfer text, numeric data other than bytes, and logical values, the SerialPort class includes methods that handle the encoding and decoding of these data types. For example, an application that wants to send the text "CMD1" followed by a LF character could store the character codes in a byte array and write the array to a port:

VB

```
Dim dataToSend(4) As Byte

dataToSend(0) = Asc("C")
dataToSend(1) = Asc("M")
dataToSend(2) = Asc("D")
dataToSend(3) = Asc("1")
dataToSend(4) = 10
myComPort.Write(dataToSend, 0, 5)
```

Table 9-2: Applications have many choices when reading from a SerialPort object.

Object	Method	Data Type Returned	Blocking
SerialPort	Read	Byte array or subarray. Char array or subarray.	Waits for at least 1 Byte or Char
	ReadByte	Byte.	yes
	ReadChar	Char.	yes
	ReadExisting	String.	no
	ReadLine	String to first NewLine. (Discards NewLine.)	yes
	ReadTo	String to first specified Char. (Discards specified Char.)	yes
BinaryReader	Read	Integer containing a character code. Byte array or subarray. Char array or subarray.	yes
	ReadBoolean	Boolean.	yes
	ReadByte	Byte.	yes
	ReadBytes	Byte array.	yes
	ReadChar	Char.	yes
	ReadChars	Char array.	yes
	ReadDecimal	Decimal.	yes
	ReadDouble	Double.	yes
	ReadInt16	Int16.	yes
	ReadInt32	Int32.	yes
	ReadInt64	Int64.	yes
	ReadSByte	SByte.	yes
	ReadSingle	Single.	yes
	ReadString	String prefixed with length.	yes
	ReadUInt16	UInt16.	yes
	ReadUInt32	UInt32.	yes
	ReadUint64	UInt64.	yes
StreamReader	Read	Integer containing a character code. Char array or subarray.	Waits for at least 1 Char
	ReadBlock	Char array or subarray.	yes
	ReadLine	String up to a NewLine. (Discards NewLine.)	yes

VC#
```
Byte[] dataToSend = new Byte[5];

dataToSend[0] = (byte)'C';
dataToSend[1] = (byte)'M';
dataToSend[2] = (byte)'D';
dataToSend[3] = (byte)'1';
dataToSend[4] = 10;

myComPort.Write(dataToSend, 0, 5);
```

But the WriteLine method offers an easier way. The method converts each character in the String to a code and writes the codes, followed by a LF character, to the port:

VB
```
dim dataToSend as String

dataToSend = "CMD1"
myComPort.WriteLine(dataToSend)
```

VC#
```
String dataToSend;

dataToSend = "CMD1";
myComPort.WriteLine(dataToSend);
```

The SerialPort class provides methods for reading and writing individual Bytes or Chars; arrays or portions of arrays of Bytes or Chars; Strings; Strings + NewLine characters; formatted Strings; Boolean values, numeric types such as Decimal, Int16, and others; and text representations of numeric values. When reading data, most methods block (don't return) until at least one byte or Char is available or a timeout occurs. The ReadExisting method is an exception that returns immediately whether data is available or not. The BytesToRead property provides a way to find out if data is available before attempting a read operation. Methods that write data also block until the write buffer is empty.

Attempting to read or write to a port that isn't open raises an InvalidOperationException. To prevent the exception, code can check to ensure the port is open and open the port if needed before reading or writing to the port.

VB
```
dim portData as String

If (Not (myComPort Is Nothing)) Then
        If (Not myComPort.IsOpen) Then
                myComPort.Open()
        End If
End If

If myComPort.IsOpen Then
        myComPort.ReadLine(portData)
        myComPort.WriteLine(portData)
End If
```

VC#
```
String portData;

if (!(ComPort.selectedPort == null))
{
        if (!myComPort.IsOpen)
{
                myComPort.Open();
        }
}
if (ComPort.selectedPort.IsOpen )
{
        portData = myComPort.ReadLine();
        myComPort.WriteLine(portData ) ;
}
```

Transferring Bytes

Many applications send and receive data stored in byte arrays and read data into byte arrays or byte variables.

Writing Bytes

The SerialPort object's Write method can write all or a portion of a byte array to a port. This example creates a 3-byte array and writes the array's contents to a port:

VB
```
Dim byteBuffer(2) As Byte

byteBuffer(0) = 117
byteBuffer(1) = 115
byteBuffer(2) = 98

myComPort.Write(byteBuffer, 0, 3)
```

VC#
```
byte[] byteBuffer = new byte[3];

byteBuffer[0] = 117;
byteBuffer[1] = 115;
byteBuffer[2] = 98;

myComPort.Write(byteBuffer, 0, 3);
```

Reading Bytes

The SerialPort class provides two methods for reading bytes.

The Read method can copy a specified number of received bytes from the receive buffer into a byte array beginning at a specified offset.

This example reads up to three received bytes, stores the bytes beginning at offset 1 in the byteBuffer array, and displays the result:

VB
```
Dim byteBuffer() As Byte = {0, 0, 0, 0}
Dim count As Integer
Dim numberOfReceivedBytes As Integer

myComPort.Read(byteBuffer, 1, 3)
For count = 0 To 3
    Console.WriteLine(CStr(byteBuffer(count)))
Next count
```

VC#
```
byte[] byteBuffer = new byte[4] {0, 0, 0, 0};
Int32 count;
Int32 numberOfReceivedBytes;

myComPort.Read(byteBuffer, 1, 3);
for (count = 0; count <= 3; count++)
{
        Console.WriteLine(byteBuffer[count].ToString() );
}
```

If the remote port sends bytes with values 10, 20, and 30, the output is this:

```
0
10
20
30
```

The Read method doesn't wait for the specified number of bytes to arrive. The method returns if there is at least one received byte in the buffer. For example, if the remote port has sent a single byte with a value of 10, the Read method in the example above returns on receiving the byte and the output is this:

```
0
10
0
0
```

Reading a byte removes the byte from the receive buffer. If no bytes arrive, the method times out as specified by the ReadTimeout property.

Another option for reading bytes is the ReadByte method, which reads one byte at a time:

VB
```
Dim receivedByte As Integer

receivedByte = myComPort.ReadByte
Console.WriteLine(receivedByte)
```

VC#
```
Int32 receivedByte;

receivedByte = myComPort.ReadByte();
Console.WriteLine(receivedByte);
```

Note that the ReadByte method reads a byte but returns an Integer with its high bytes set to zero. If no byte is available, the method waits up to the time specified in the ReceiveTimeout property.

Transferring Text

Chapter 2 introduced methods of encoding text. The SerialPort class provides a variety of methods for reading and writing text data as Strings and Char variables.

Writing Text

The Write and WriteLine methods can write text to a port.

The Write method used for writing byte arrays can also write Strings:

VB
```
Dim textToWrite As String

textToWrite = "hello, world"
myComPort.Write(textToWrite)
```

VC#
```
String textToWrite;

textToWrite = "hello, world";
myComPort.Write(textToWrite);
```

The WriteLine method is similar to Write but appends a NewLine character or characters to the String. The default NewLine value is a line feed (LF, or code point U+000A). Other possible NewLine values are a carriage return (CR, or U+000D), or a CR followed by a LF. Visual Basic defines constants for these values (VbLf, VbCr, and VbCrLf).

The carriage-return and line-feed characters affect how text is printed or displayed on the receiving computer. To save bandwidth, software at a receiving computer can convert each received LF into a CR+LF if needed.

The Write method can also write all or a portion of a Char array to a port. This example places two characters in an array and writes the characters to a port:

VB
```
Dim textToWrite(1) As Char

textToWrite(0) = "h"c
textToWrite(1) = "i"c
myComPort.Write(textToWrite, 0, 2)
```

VC#
```
Char[] textToWrite = new Char[2];

textToWrite[0] = 'h' ;
textToWrite[1] = 'i';
myComPort.Write(textToWrite, 0, 2);
```

Reading Text

The SerialPort class provides five methods for reading text.

The ReadExisting method returns a String containing all of the characters in the receive buffer. If the buffer is empty, the method returns an empty String. ReadExisting is the only read method that doesn't block until receiving data or a timeout.

VB
```
Dim receivedText as String

receivedText = myComPort.ReadExisting
```

VC#
```
String receivedText;

receivedText = myComPort.ReadExisting();
```

The ReadLine method returns all received data up to a NewLine character:

VB
```
Dim receivedText as String

receivedText = myComPort.ReadLine
```

VC#
```
String receivedText;

receivedText = myComPort.ReadLine();
```

The returned String contains everything up to the NewLine. The NewLine character(s) are removed from the receive buffer and discarded. If the buffer doesn't contain a NewLine, the method waits up to the ReadTimeout value and then raises a timeout exception.

The ReadTo method is like ReadLine but enables defining any value as the delimiter.

The Read method can copy a specified number of received Chars into a Char array starting at a specified offset.

This example reads up to three received Chars, stores the bytes beginning at offset zero in a byte array, and displays the result:

VB

```
Dim charBuffer(7) As Char
Dim count As Integer

myComPort.Read(charBuffer, 0, 3)

For count = 0 To 2
        Console.WriteLine(CStr(charBuffer(count)))
Next count
```

VC#

```
char[] charBuffer = new char[8];
int count = 0;
myComPort.Read(charBuffer, 0, 3)

for ( count=0; count <= 2; count++ )
{
        Console.WriteLine(System.Convert.ToString(charBuffer[count]));
}
```

If the buffer contains the characters "a", "b", and "c", the output is this:

```
Number of received bytes = 3
a
b
c
```

Characters that have been read are no longer available in the receive buffer. When reading a byte array, the Read method stores one byte in each array element. When reading a Char array, the Read method stores received data as Unicode characters. Each element in a Char array represents one character, whether the character's encoding uses one or more bytes. The Read method returns when at least one character is available or on a timeout.

The ReadChar method reads a single Char from the receive buffer. The method returns an integer that contains an encoded character.

VB

```
Dim charToRead As Integer

charToRead = myComPort.ReadChar()
Console.WriteLine(ChrW(charToRead))
```

VC#
```
int charToRead;

charToRead = ComPort.selectedPort.ReadChar();
Console.WriteLine((char)(charToRead));
```

Character Encoding

The .NET Framework defaults to UTF-16 encoding for most strings and characters, but the default encoding for SerialPort objects is ASCIIEncoding. With ASCIIEncoding, character codes in the range 00h–7Fh are identical to Unicode UTF-8 characters, and all character codes from 80h to FFh transmit as question marks (3Fh).

This example creates a String that contains five 16-bit characters and writes five bytes (an 8-bit code for each character) to the SerialPort object:

VB
```
Dim textToSend As String = "hello"

Dim ascii As New System.Text.ASCIIEncoding()

myComPort.Encoding = ascii
myComPort.Write(textToSend)
```

VC#
```
String textToSend = "hello";

System.Text.ASCIIEncoding ascii = new System.Text.ASCIIEncoding();

myComPort.Encoding = ascii;
myComPort.Write(textToSend);
```

The transmitting computer stores the character "h" as a 16-bit value (0068h) in the String textToSend. The Write method causes the character "h" to transmit on the port as the low byte of the encoded character (68h). The other character codes also transmit as single bytes. ASCIIEncoding is the default encoding, so explicitly assigning ASCIIEncoding to the Encoding property is optional.

If a SerialPort object is using ASCIIEncoding and storing received data in a String or Char array, each received byte is stored by default as a 16-bit character:

VB

```
' Read a String.

Dim receivedText1 As String

receivedText1 = myComPort.ReadExisting
Console.WriteLine(receivedText1)

' Read a Char array.

Dim receivedText2(3) As Char

myComPort.Read(receivedText2, 0, 4)
Console.WriteLine(receivedText2)
```

VC#

```
// Read a String.

String receivedText1;

receivedText1=myComPort.ReadExisting();
Console.WriteLine(receivedText1);

// Read a Char array.

Char[] receivedText2 = new Char[4];

myComPort.Read(receivedText2, 0, 4);
Console.WriteLine(receivedText2);
```

The ReadExisting and Read methods retrieve received characters and store each as a 16-bit value in a String.

The transmitting and receiving computers must agree on an encoding method for the text. For example, in the code below, the transmitting computer might think it's transmitting a byte equal to 169 (A9h), which is the code point for a copyright symbol (©):

VB

```
Dim charToSend as Char = ChrW(169)

myComPort.Write(charToSend)
```

VC#
```
Char charToSend = (char)169;

myComPort.Write(System.Convert.ToString(charToSend));
```

But if the transmitting port is using ASCIIEncoding, the port transmits 3Fh. The receiving computer reads the character:

VB
```
Dim receivedChar As Char

receivedChar = ChrW(myComPort.ReadChar)
Console.WriteLine(receivedChar)
```

VC#
```
Int32 receivedChar;

receivedChar = (myComPort.ReadChar());
Console.WriteLine(System.Convert.ToChar(receivedChar));
```

And the console output is:

```
?
```

To enable transmitting characters with character codes 80h–FFh, an application can declare a different encoding. One option is UTF-8 encoding:

VB
```
Dim utf8 As New System.Text.UTF8Encoding()

myComPort.Encoding = utf8
```

VC#
```
System.Text.UTF8Encoding utf8 = new System.Text.UTF8Encoding();

myComPort.Encoding = utf8;
```

With this encoding, the transmitted bytes don't always match the code points of the characters because UTF-8 encoding converts code points 80h–FFh to multiple 8-bit code units. If a computer using UTF-8 encoding communicates with a computer using ASCIIEncoding or another encoding, the receiving computer won't decode the data correctly.

If the transmitting and receiving computers both use UTF8Encoding and run the example code above, the transmitting computer converts the UTF-16 string 00A9h to the UTF-8 code units C2h A9h and transmits the two bytes on the serial port. The receiving computer converts the received code units to the UTF-16 String 00A9h and displays:

```
©
```

A receiving computer using ASCIIEncoding will display "?" for each received byte greater than 7Fh received:

??

Another option is to use UTF-16 and transmit and receive all characters as 16-bit code units:

VB
```
Dim utf16 As New System.Text.UnicodeEncoding()
myComPort.Encoding = utf16
```

VC#
```
System.Text.UnicodeEncoding utf16 = new System.Text.UnicodeEncoding();

myComPort.Encoding = utf16;
```

To send the copyright symbol, a transmitting computer using UTF-16 encoding sends the bytes A9h 00h, and the receiving computer stores the bytes as the character 00A9h. The down side is that data takes twice as long to transmit and a receiving computer that uses 8-bit characters will have to convert the received data. Note that the UTF-16 encoding property is called UnicodeEncoding (not UTF16Encoding).

Using Stream Objects

Stream objects can represent a source or destination for a series of bytes. The source or destination can be a storage medium such as a file, memory, or a connection to a device such as a COM port (including virtual COM ports) or TCP/IP socket. The Stream class provides methods for reading and writing to stream objects.

A generic Stream object can read and write bytes. Objects created using auxiliary stream classes support reading and writing bytes as well as other data types. SerialPort objects can use the auxialiary BinaryReader and BinaryWriter classes to read and write numeric data and text and the auxiliary StreamReader and StreamWriter classes to read and write text. Closing a SerialPort object disposes of the port's Stream object.

The stream classes are in the System.IO namespace:

VB
```
Imports System.IO
```

VC#
```
using System.IO;
```

BinaryReader and BinaryWriter

The BinaryReader and BinaryWriter classes support reading and writing data in a variety of formats, including bytes and other numeric types, Boolean data, Chars, and length-prefixed strings.

An application uses the SerialPort object's BaseStream property to obtain a Stream object and then uses the object to create BinaryReader and BinaryWriter objects:

VB

```
Friend binaryReader1 As BinaryReader
Friend binaryWriter1 As BinaryWriter
Dim serialPortStream As Stream

' Get the port's Stream object.

serialPortStream = myComPort.BaseStream

' Use the Stream object to create BinaryReader and BinaryWriter objects.

binaryReader1 = New BinaryReader(serialPortStream)
binaryWriter1 = New BinaryWriter(serialPortStream)
```

VC#

```
internal BinaryReader binaryReader1;
internal BinaryWriter binaryWriter1;
Stream serialPortStream = null;

// Get the port's Stream object.

serialPortStream = myComPort.BaseStream;

// Use the Stream object to create BinaryReader and BinaryWriter objects.

binaryReader1 = new BinaryReader( serialPortStream );
binaryWriter1 = new BinaryWriter( serialPortStream );
```

The Write method can write a byte to a port:

VB

```
Dim dataToWrite As Byte

dataToWrite = 65
binaryWriter1.Write(dataToWrite)
```

VC#
```
byte dataToWrite;

dataToWrite = 65;
binaryWriter1.Write( dataToWrite );
```

The Read method can read a received byte into a byte array:

VB
```
Dim receivedData(0) As Byte

binaryReader1.Read(receivedData, 0, 1)
Console.WriteLine(ChrW(receivedData(0)))
```

VC#
```
byte[] receivedData = new byte[ 1 ];

binaryReader1.Read( receivedData, 0, 1 );
Console.WriteLine((char)( receivedData[ 0 ] ) );
```

The Write method can also write just about any data type. For example, a BinaryWriter object can write an Int32 value:

VB
```
Dim valueToWrite As Int32

valueToWrite = 66
binaryWriter1.Write(valueToWrite)
```

VC#
```
Int32 dataToWrite;

dataToWrite = 66;
binaryWriter1.Write( dataToWrite );
```

Each Int32 value written to the port is four bytes. For example, writing an Int32 with a value of 66 (42h) writes the bytes 42h, 00h, 00h, 00h to the port.

The receiving computer can store the value in an Int32 variable:

VB
```
Dim receivedData As Int32

receivedData = binaryReader1.ReadInt32
```

VC#
```
Int32 receivedData;

receivedData = binaryReader1.ReadInt32();
```

The ReadInt32 method returns on receiving four bytes.

The Write and Read methods can also write and read Chars and Char arrays.

VB

```
‘ Write a Char.

Dim valueToWrite As Char

valueToWrite = CChar("A")
binaryWriter1.Write(valueToWrite)

' Read a Char.

Dim receivedData As Integer

receivedData = binaryReader1.Read
Console.WriteLine(ChrW(receivedData))

' Write an array of Chars.

Dim valueToWrite(1) As Char

valueToWrite(0) = CChar("O")
valueToWrite(1) = CChar("K")

binaryWriter1.Write(valueToWrite, 0, 2)

' Read Chars into an array.

Dim receivedData(1) As Char

binaryReader1.Read(receivedData, 0, 2)
Console.WriteLine(receivedData)
```

VC#
```
// Write a Char.

 Char valueToWrite;

valueToWrite = 'A';
binaryWriter1.Write( valueToWrite );

// Read a Char.

int receivedData;

receivedData = binaryReader1.Read();
Console.WriteLine( Strings.ChrW( receivedData ) );

// Write an array of Chars.

Char[] valueToWrite = new Char[2];

valueToWrite[0] = 'O';
valueToWrite[1] = 'K';
binaryWriter1.Write( valueToWrite,0,2 );

// Read Chars into an array.

Char[] receivedData = new Char[2];

binaryReader1.Read(receivedData,0,2);

Console.WriteLine(receivedData);
```

BinaryWriter and BinaryReader can also send and receive strings. Each string is prefixed with a length value. Before transmitting a string, a BinaryWriter object encodes a 32-bit length value using the Write7BitEncodedInt method. The method encodes the length value as one or more bytes depending on the value being encoded.

You don't have to understand the encoding method to use it in an application. BinaryReader and BinaryWriter create and extract the length values automatically. With that said, this is how the encoding works:

The encoding divides the value to transmit into units of 7 bits each. Each unit is stored in bits 0–6 of a prefix byte to be transmitted. In the byte that contains the most significant bits of the length value, bit 7 is zero. For all of the other bytes, bit 7 is set to 1.

For example, a 5-byte string has a 1-byte prefix of 05h. A 128-byte string has a 2-byte prefix of 80h 01h. The length to be encoded (128) equals 10000000_b. The length value has 8 bits so the prefix is 2 bytes. In the prefix's first byte (80h), bits 0–6 match bits 0–6 in the length value and bit 7 = 1 to indicate there is another prefix byte. In the prefix's second byte (01h), bit 0 matches bit 7 in the length value, bits 1–6 are zeroes, and bit 7 equals zero to indicate that the current byte is the last byte in the prefix.

The value being encoded is the number of bytes transmitted, not the number of characters, and excludes the prefix. When a BinaryWriter sends a © character by writing the bytes C2h A9h, the prefix is 2 to indicate 2 bytes being transmitted.

To write a String with BinaryWriter, use the Write method:

VB

```
Dim valueToWrite As String

valueToWrite = "hello"
binaryWriter1.Write(valueToWrite)
```

VC#

```
String valueToWrite ;

valueToWrite =  "hello";
binaryWriter1.Write( valueToWrite );
```

To read a String with BinaryReader, use the ReadString method:

VB

```
Dim receivedData As String

receivedData = binaryReader1.ReadString
Console.WriteLine(receivedData)
```

VC#

```
String receivedData;

receivedData = binaryReader1.ReadString();
Console.WriteLine(receivedData):
```

The ReadString method uses the length prefix to determine the string's length, discards the prefix, and returns the string that follows. If transmitted data doesn't include a length prefix, the receiving computer interprets the initial byte(s) as a length value instead of text data and thus doesn't read the correct string and raises a TimeoutException if the expected number of bytes doesn't arrive.

The default text encoding for BinaryReader and BinaryWriter is UTF8. Constructors can specify a different encoding method for the streams:

VB
```
Friend binaryReader2 As BinaryReader
Friend binaryWriter2 As BinaryWriter
Dim serialPortStream As Stream
Dim utf16 As New System.Text.UnicodeEncoding

' Get the port's Stream object.

serialPortStream = myComPort.BaseStream

' Use the Stream object to create BinaryReader and BinaryWriter objects.

binaryReader2 = New BinaryReader(serialPortStream, utf16)
binaryWriter2 = New BinaryWriter(serialPortStream, utf16)
```

VC#
```
internal BinaryReader binaryReader2;
internal BinaryWriter binaryWriter2;
System.Text.UnicodeEncoding utf16 = new System.Text.UnicodeEncoding();

 Stream serialPortStream = null;

//  Get the port's Stream object.

serialPortStream = myComPort.BaseStream;

//  Use the Stream object to create BinaryReader and BinaryWriter objects.

binaryReader2 = new BinaryReader( serialPortStream,utf16 );
binaryWriter2 = new BinaryWriter( serialPortStream,utf16 );
```

The PeekChar method, which returns the next value in a stream without removing the value from the stream, isn't supported by SerialPort BinaryReader objects.

StreamReader and StreamWriter

The StreamReader and StreamWriter classes support reading and writing text using a specified Unicode encoding. The object's Encoding property determines the encoding method. The default encoding is UTF-8. Unlike BinaryReader and BinaryWriter, StreamReader and StreamWriter don't use length prefixes.

As with BinaryReader and BinaryWriter objects, an application can create StreamReader and StreamWriter objects using the SerialPort object's BaseStream property:

VB
```
serialPortStream = myComPort.BaseStream
streamReader1 = New StreamReader(serialPortStream)
streamWriter1 = New StreamWriter(serialPortStream)
```

VC#
```
serialPortStream = myComPort.BaseStream;
streamReader1 = new StreamReader(serialPortStream);
streamWriter1 = new StreamWriter(serialPortStream);
```

Setting StreamWriter's AutoFlush property to True causes Write operations to send data immediately to the port:

VB
```
streamWriter1.AutoFlush = True
```

VC#
```
streamWriter1.AutoFlush = true;
```

If the AutoFlush property is False, data written to the stream may be delayed in the stream's buffer. When AutoFlush is False, calling the Flush method or closing the stream sends any buffered data to the port.

The Write and WriteLine methods can write Strings to a port:

VB
```
streamWriter1.Write("hello, ")
streamWriter1.WriteLine("world")
```

VC#
```
streamWriter1.Write("hello, ");
streamWriter1.WriteLine("world");
```

In addition to basic Strings, StreamWriter's Write and WriteLine methods can write Chars, text representations of numeric and Boolean values, text representations of objects as defined an object's ToString property, and strings formatted as defined by the String.Format property.

Here are some examples:

VB
```
Dim dataToSend1 As Char = CChar("A")
streamWriter1.WriteLine(dataToSend1)

Dim dataToSend2 As Int32 = 254
streamWriter1.WriteLine(dataToSend2)

Dim dataToSend3 As Decimal = 100
streamWriter1.WriteLine(dataToSend3)

Dim dataToSend4 As Boolean = True
streamWriter1.WriteLine(dataToSend4)

Dim dataToSend5 As Byte = 57
streamWriter1.WriteLine(dataToSend5.ToString)

streamWriter1.WriteLine("{0:t}", DateTime.Now)
```

VC#
```
Char dataToSend1 = 'A';
streamWriter1.WriteLine(dataToSend1);

Int32 dataToSend2 = 254;
streamWriter1.WriteLine(dataToSend2);

Decimal dataToSend3 = 100;
streamWriter1.WriteLine(dataToSend3);

Boolean dataToSend4 = true;
streamWriter1.WriteLine(dataToSend4);

Byte dataToSend5  = 57;
streamWriter1.WriteLine(dataToSend5.ToString ());

streamWriter1.WriteLine("{0:t}", DateTime.Now);
```

The code above writes the following text to the stream and port:

```
A
254
100
True
57
4:07 PM
```

The StreamReader class provides several methods for reading text from a SerialPort object.

StreamReader's Read method can read a single character into an Integer or one or more characters into a Char array. The method returns when at least one character has been received:

VB
```
' Read a character.

Dim receivedData1 As Integer

receivedData1 = streamReader1.Read
Console.WriteLine(Chr(receivedData1))

'  Read two characters into an array.

Dim receivedData2(1) As Char
Dim count as Integer

count = streamReader1.Read(receivedData2, 0, 2)
Console.WriteLine(count & " characters received: ")
Console.WriteLine(receivedData2)
```

VC#
```
// Read a character.

int receivedData1;

receivedData1 = streamReader1.Read();
Console.WriteLine(Strings.ChrW(receivedData1));

// Read two characters into an array.

Char[] receivedData2 = new Char[2];
int count;

count = streamReader1.Read(receivedData2, 0, 2);
Console.WriteLine(count + " characters received: ");
Console.WriteLine(receivedData2);
```

ReadBlock is a blocking version of the Read method and stores received data in a Char array. Unlike Read, ReadBlock waits for all of the requested characters to arrive or a timeout.

ReadLine returns a String containing everything up to the first NewLine character.

Peek and ReadToEnd are methods that might appear useful but in reality aren't practical for most COM-port applications.

Peek returns the next value in the stream without removing the value from the stream or returns -1 if no character is available. The method can be useful when reading from resources such as files but is less helpful in COM-port applications that send data continuously. When the buffer is empty, Peek returns -1 but then continues to return -1 even after more characters have arrived at the port.

ReadToEnd returns a String containing everything from the current position to the end of the stream. The stream normally has no way of knowing when it has reached the end of a particular block of COM-port data. (When reading from a StreamWriter object, ReadToEnd returns when the StreamWriter object closes.)

Saving a Port and Parameters

Users often find it convenient when an application remembers and selects the port used the last time the application ran. One way to save parameters is to store them in the system registry using the RegistryKey class in the Microsoft.Win32 namespace. The .NET Framework provides another option in the Application Settings architecture, which enables storing and retrieving settings in files that don't clog the registry.

The Application Settings Architecture

The System.Configuration namespace includes the AppSettingsSection class, which provides the Settings property for saving and retrieving settings. The Settings property is a NameValueCollection of strings and string values. (The NameValueCollection class is in the System.Collections.Specialized namespace.) The Application Settings architecture was introduced in .NET 2.0.

To create a setting, in Visual Studio, in the Project menu, select *Project_name* Properties, where *Project_name* is the name of your project. In the window that appears, select Settings (Figure 9-1). For each setting, enter a name and default value and select a data type and scope. User scope is for settings that users can change, while application scope is generally for settings that never change.



Figure 9-1: In Visual Studio's Project > Properties windows, the Settings pane enables defining application-specific values to be saved with an application.

The settings are stored in the file *user.config*, where *user* is the user name of the person running the application. To enable using the settings, you must deploy Visual Studio's *app.exe* file with the project.

Combo Box Example

An application might save its settings when executing the main form's _FormClosing routine. This example saves the selected items in three combo boxes (cmbBitRate, cmbPort, and cmbHandshaking):

VB

```
Settings.Default.BitRate = CInt(cmbBitRate.SelectedItem)
Settings.Default.ComPort = cmbPort.SelectedItem.ToString
Settings.Default.Handshaking = DirectCast(cmbHandshaking.SelectedItem, Handshake)

Settings.Default.Save()
```

VC#

```
// COMPortTerminal is the project's default namespace
// (specified in Project > Properties > Application).

using COMPortTerminal.Properties;

Settings.Default.BitRate = (int)cmbBitRate.SelectedItem;
Settings.Default.ComPort = cmbPort.SelectedItem.ToString();
Settings.Default.Handshaking = (Handshake)cmbHandshaking.SelectedItem;

Settings.Default.Save();
```

On running the application, the main form's _Load routine can retreive the settings:

VB

```
cmbBitRate.SelectedItem = Settings.Default.BitRate
cmbHandshaking.SelectedItem = Settings.Default.Handshaking
cmbPort.SelectedItem = Settings.Default.ComPort
```

VC#

```
cmbBitRate.SelectedItem = Settings.Default.BitRate;
cmbHandshaking.SelectedItem = Settings.Default.Handshaking;
cmbPort.SelectedItem = Settings.Default.ComPort;
```

USB virtual COM ports can come and go on a PC as users attach and remove devices. A retrieved port name might specify a port that isn't available. Before attempting to open a default port, application code can search the array returned by GetPortNames to determine if the port exists, as described earlier in this chapter.

10

Managing Ports and Transfers in .NET

The .NET Framework and its SerialPort class provide many properties, methods, and events that are useful in managing transfers. Applications can define buffers to hold data, implement flow control to prevent lost data, and use timers or DataReceived events to detect received data. Parity bits can enable detecting errors in received data. This chapter demonstrates how to use these capabilities to transfer data efficiently and without errors.

The code in this chapter uses resources in the following namespaces:

VB

```
Imports System.IO.Ports
Imports System.Runtime.Remoting.Messaging  ' used with asynchronous delegates
```

VC#

```
using System.IO.Ports;
using System.Runtime.Remoting.Messaging;  // used with asynchronous delegates
```

Several of the examples in this chapter use delegates to perform event-driven or asynchronous (non-blocking) operations. A delegate is an object that defines the number and type(s) of parameters a method accepts and the type of value

the method returns, if any. A delegate enables application code to invoke a method indirectly and safely. The examples use delegates in several ways. Delegates specify routines to handle events that receive data, detect serial-port errors, and detect changes on RS-232 status pins. Delegates also enable passing received data to a form and performing write operations without blocking user input or other operations performed by the application's main thread.

Receiving Data

Many applications need to detect COM-port data that arrives at unpredictable times. Some applications need to collect and manage data that arrives over a period of time. This section shows strategies for accomplishing these tasks.

Setting Timeouts

The ReadTimeout property introduced in Chapter 9 can keep a program thread from wasting too much time waiting for data to arrive. When ReadTimeout is greater than zero, all of the read methods except ReadExisting will block the thread that called the method until data is available or a timeout. A thread that calls a blocking method can do no other work, including responding to user input, until the method returns.

On a timeout when transmitting data, the application might want to set a variable that tells the application to retry. On a timeout when waiting to receive expected data, the application might want to send a message to notify the transmitting computer of the problem.

Detecting Received Data

To detect received data, applications can poll the port or use the DataReceived event.

Polling

An application can poll a port at specific times when the application expects to receive data or at defined intervals. If no data is available, a blocking read method will wait for data or a timeout.

To avoid waits and timeouts, an application can read the BytesToRead property and attempt to read data from the port only if data is available:

VB

```
Dim receivedData As Integer

If myComPort.BytesToRead > 0 Then
        receivedData = myComPort.ReadByte
        Console.WriteLine(ChrW(receivedData))
Else
        Console.WriteLine("No data available.")
End If
```

VC#

```
int receivedData;

if (myComPort.BytesToRead > 0)
{
        receivedData = myComPort.ReadByte();
        Console.WriteLine((char)receivedData);
}
else
{
        Console.WriteLine("No data available.");
}
```

If using a method such as BinaryReader's ReadInt32, which reads multi-byte values, be sure to check for the needed number of bytes in the buffer.

For reading strings, the ReadExisting method always returns immediately, returning an empty string if the buffer is empty. For the blocking read methods, setting the ReadTimeout property determines how long the method waits if no data is available.

An example of an application that expects to receive data at specific times is a computer that sends commands and expects a response after each command. If the remote computer typically responds quickly, the application might wait for the response:

VB
```
Dim receivedData As String

' Send a command.

myComPort.WriteLine("CMD1")

' Wait for a response.

receivedData = myComPort.ReadLine
Console.WriteLine(receivedData)
```

VC#
```
String receivedData;

// Send a command.

myComPort.WriteLine("CMD1");

// Wait for a response.

receivedData = myComPort.ReadLine();
Console.WriteLine(receivedData);
```

Another option for polling is to use a Timer component to trigger periodic reads of the port. Drag a Timer component from Visual Studio's Toolbox onto a form. Set the interval in milliseconds for polling the port.

VB
```
tmrPollForReceivedData.Interval = 1000
tmrPollForReceivedData.Stop()
```

VC#
```
tmrPollForReceivedData.Interval = 1000;
tmrPollForReceivedData.Stop();
```

When the port has been opened, start the timer:

VB
```
tmrPollForReceivedData.Start()
```

VC#
```
tmrPollForReceivedData.Start();
```

The timer's Tick routine executes when the timer is running and the specified interval has elapsed. The routine should do whatever is needed with any received data:

VB
```
Private Sub tmrPollForReceivedData_Tick _
        (ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles tmrPollForReceivedData.Tick

        Dim receivedData As String

        ' Read and display any received data.

        receivedData = myComPort.ReadExisting()

        If Not (receivedData = "") Then
                Console.WriteLine(receivedData)
        End If

End Sub
```

VC#
```
private void tmrPollForReceivedData_Tick(object sender, EventArgs e)
{
        String receivedData;

        // Read and display any received data.

        receivedData = myComPort.ReadExisting();

        if (!(receivedData == "") )
        {
                Console.WriteLine(receivedData);
        }
}
```

On closing the port, stop the timer.

VB
```
tmrPollForReceivedData.Stop()
```

VC#
```
tmrPollForReceivedData.Stop();
```

Using the DataReceived Event

The DataReceived event provides an efficient way to detect received data. To use the event, create a delegate for a routine that will execute when a DataReceived event occurs on an open port.

VB
```
' Define a delegate class to handle DataReceived events.

Friend Delegate Sub SerialDataReceivedEventHandlerDelegate _
        (ByVal sender As Object, ByVal e As SerialDataReceivedEventArgs)

' Create an instance of the delegate.

 Private SerialDataReceivedEventHandler1 _
       As New SerialDataReceivedEventHandler(AddressOf DataReceived)
```

VC#
```
// Define a delegate class to handle DataReceived events.

internal delegate void SerialDataReceivedEventHandlerDelegate
        (object sender, SerialDataReceivedEventArgs e);

// Create an instance of the delegate.

private static SerialDataReceivedEventHandler SerialDataReceivedEventHandler1 =
        new SerialDataReceivedEventHandler( ComPort.DataReceived );
```

Assign the delegate as the handler routine to execute when a DataReceived event occurs on a port:

VB
```
AddHandler myComPort.DataReceived, SerialDataReceivedEventHandler1
```

VC#
```
myComPort.DataReceived += SerialDataReceivedEventHandler1;
```

The DataReceived routine can then read and process received data as needed.

VB
```
Friend Sub DataReceived _
        (ByVal sender As Object, ByVal e As SerialDataReceivedEventArgs)

        ' Place code to read and process data here.

End Sub
```

VC#
```
internal void DataReceived( object sender, SerialDataReceivedEventArgs e )
{
        ' Place code to read and process data here.
}
```

The code in the routine can be identical to the code in the tmrPollForReceivedData_Tick routine above or whatever the application requires. The ReceivedBytesThreshold property specifies how many bytes must be available before the event is raised.

Marshaling Received Data to a Form

The DataReceived routine executes in a secondary thread. To display data, messages, or other information from the DataReceived routine on an application's form, an application must marshal the data from the DataReceived routine to the form.

In the example below, the DataReceived routine reads data from the port and passes the data in a call to AccessFormMarshal. The AccessFormMarshal routine uses a delegate to pass the received data to the AccessForm routine, which displays the received data on the form.

To enable marshaling data, the form's module defines a delegate class with parameters that will hold the data to be passed to the form. This example passes a string to display:

VB
```
' MyMainForm is an instance of the form MainForm.

Friend MyMainForm As MainForm

Private Delegate Sub AccessFormMarshalDelegate(ByVal textToDisplay As String)
```

VC#
```
// MyMainForm is an instance of the form MainForm.

internal static MainForm MyMainForm;

private delegate void AccessFormMarshalDelegate(string textToDisplay);
```

Create a routine with the same parameter(s) as the delegate:

VB
```
Private Sub AccessFormMarshal(ByVal textToDisplay As String)

    ' The parameter(s) to pass to the form.

    Dim args() As Object = {textToDisplay}

    ' The AccessForm routine contains the code that accesses the form.

    Dim AccessFormMarshalDelegate1 As _
        New AccessFormMarshalDelegate(AddressOf AccessForm)

    ' Call AccessForm, passing the parameters in args.

    MyBase.Invoke(AccessFormMarshalDelegate1, args)

End Sub
```

VC#
```
internal void AccessFormMarshal(string textToDisplay)
{
    // The parameter(s) to pass to the form.

    object[] args = {textToDisplay};

    AccessFormMarshalDelegate AccessFormMarshalDelegate1 = null;

    //  The AccessForm routine contains the code that accesses the form.

    AccessFormMarshalDelegate1 = new AccessFormMarshalDelegate(AccessForm);

    //  Call AccessForm, passing the parameters in args.

    base.Invoke(AccessFormMarshalDelegate1, args);
}
```

Create the routine that accesses the form. This example appends the passed text to a text box (txtUserDisplay) on the form:

VB
```
Private Sub AccessForm(ByVal textToDisplay As String)

    txtUserDisplay.AppendText(textToDisplay)
End Sub
```

VC#
```
private void AccessForm(string textToDisplay)
{
        txtUserDisplay.AppendText(textToDisplay);
}
```

In the DataReceived routine, call AccessFormMarshal to pass data to display in the form's textbox:

VB
```
Dim receivedData As String = myComPort.ReadExisting()

MyMainForm.AccessFormMarshal(receivedData)
```

VC#
```
string receivedData = myComPort.ReadExisting();

MyMainForm.AccessFormMarshal(receivedData);
```

Collecting Received Data

Many applications collect a quantity of received data before taking action. For example, a remote computer might send a large file for the receiving computer to store, execute, or use in another way. A transmitting computer might also send large quantities of data in multiple files or other blocks of data. In either case, the application may need to provide a location to store data in sequence from multiple read operations.

The ReadBufferSize property sets the size of the receive buffer with a minimum size of 4096 bytes. If you expect a port to collect more than 4096 bytes of received data before the application can read the data, increase the size of the buffer. An application that knows how many bytes to expect can let the data collect in the SerialPort buffer and use the BytesReceived property to find out when all of the data has arrived.

Sometimes an application doesn't know how much data will arrive, or the application might need to examine the data as it comes in. The StringBuilder and List(Of T) classes provide mechanisms for collecting and managing received data.

Using the StringBuilder Class

The StringBuilder class can collect received strings in a buffer that grows as needed when the application appends new strings. For example, if receiving a text file, an application can store the file as a single StringBuilder object. The class is in the System.Text namespace.

VB
```
Imports System.Text

Friend stringBuffer As New StringBuilder

Dim newReceivedData As String

' Read received data into a String.

newReceivedData = myComPort.ReadExisting()

' Append the String to the StringBuilder object.

stringBuffer.Append(newReceivedData)
```

VC#
```
using System.Text;

internal StringBuilder stringBuffer = new StringBuilder();

String newReceivedData;

// Read received data into a String.

newReceivedData = myComPort.ReadExisting();

// Append the String to the StringBuilder object.

stringBuffer.Append(newReceivedData);
```

To convert a StringBuilder object to a String, use the ToString property:

VB
```
Dim receivedText as String

receivedText = stringBuffer.ToString
```

VC#
```
String receivedText;

receivedText = stringBuffer.ToString();
```

To clear a StringBuilder object, use the Remove method:

VB
```
stringBuffer.Remove(0, stringBuffer.Length)
```

VC#
```
stringBuffer.Remove(0, stringBuffer.Length);
```

The Remove method can also remove selected characters:

VB
```
' Remove the first two characters:

stringBuffer.Remove(0, 2)
```

VC#
```
// Remove the first two characters:

stringBuffer.Remove(0, 2);
```

You can specify a capacity when declaring a StringBuilder object:

VB
```
Friend stringBuffer As New StringBuilder(1024)
```

VC#
```
internal StringBuilder stringBuffer = new StringBuilder(1024);
```

The StringBuilder object can store up to the specified capacity without having to allocate more memory. On exceeding the specified capacity, the object grows in size as needed to hold the data.

Using the List(Of T) Class

For storing bytes and other data types as well as strings, the List(Of T) class (the List class in C#) offers a solution. The class is in the System.Collections.Generic namespace and supports methods for manipulating, searching, and sorting list elements. The AddRange method can add the contents of an array to the end of a list, expanding the list's capacity as needed. The RemoveRange method can remove a range of elements from a list. The Clear method removes all elements in a list.

A List(Of T) object can hold a series of bytes:

VB
```
Imports System.Collections.Generic

Friend portBuffer As New List(Of Byte)
```

VC#
```
using System.Collections.Generic;

internal List<Byte> portBuffer = new List<Byte>();
```

To reduce the number of resizing operations needed as elements are added to the list, you can specify an initial capacity when creating the list object:

VB
```
Friend portBuffer As New List(Of Byte)(1024)
```

VC#
```
internal List<Byte> portBuffer = new List<Byte>(1024);
```

The code below appends received bytes to the end of a list and calls a routine to process the data. The code can execute in a DataReceived event or Timer event:

VB
```
Dim numberOfBytesToRead As Integer

' Get the number of bytes available to read.

numberOfBytesToRead = myComPort.BytesToRead

' Create a byte array large enough to hold the bytes to be read.

Dim newReceivedData(numberOfBytesToRead - 1) As Byte

' Read the bytes into the byte array.

myComPort.Read(newReceivedData, 0, numberOfBytesToRead)

' Add the bytes to the end of the list.

portBuffer.AddRange(newReceivedData)

' Call a routine to process the data.

ProcessData()
```

VC#
```
int numberOfBytesToRead;

// Get the number of bytes available to read.

numberOfBytesToRead = myComPort.BytesToRead;

// Create a byte array large enough to hold the bytes to be read.

Byte[] newReceivedData = new Byte[numberOfBytesToRead];

// Read the bytes into the byte array.

myComPort.Read(newReceivedData, 0, numberOfBytesToRead);

// Add the bytes to the end of the list.

portBuffer.AddRange(newReceivedData);

// Call a routine to process the data.

ProcessData();
```

The routine below illustrates how to use a List(Of T) object to collect data. The routine waits for eight bytes to arrive, removes the bytes from the list, and displays the bytes as text characters.

VB
```
Private Sub ProcessData()

    ' When eight bytes have arrived, display them and remove them from the buffer.

    Dim count As Integer
    Dim numberOfBytesToRead As Integer = 8

    If portBuffer.Count >= numberOfBytesToRead Then

        For count = 0 To numberOfBytesToRead - 1
            Console.WriteLine(ChrW(portBuffer(count)))
        Next count

        ' Remove the bytes read from the list.
        portBuffer.RemoveRange(0, numberOfBytesToRead)
    End If
End Sub
```

VC#
```
private void ProcessData()
{
    // When eight bytes have arrived, display them and remove them from the buffer.

    int count;
    int numberOfBytesToRead = 8;

    if (portBuffer.Count >= numberOfBytesToRead)
    {
        for (count = 0; count < numberOfBytesToRead; count++)
        {
            Console.WriteLine((char)(portBuffer[count]));
        }
        // Remove the bytes read from the list.

        portBuffer.RemoveRange(0, numberOfBytesToRead);
    }
}
```

In a similar way, an application can create a list of Chars or Strings for storing received text. To add a received string to a list, use the Add method.

Ensuring Efficient Transfers

Each read operation adds overhead, which can degrade performance of applications that are reading large amounts of data. For example, an application reading a large file from a port might want to read the file using as few read operations as possible rather than collecting each byte as it arrives. An application can read a port's BytesToRead property to find out if the needed number of bytes is available before reading from the port.

When using the ReadExisting method, call the method less often to retrieve more data on each call. When using the DataReceived event, set ReceivedBytesThreshold to a large block size or to the expected number of bytes if known. A timer can trigger a read of any amount smaller than ReceivedBytesThreshold that remains unread in the buffer. When waiting for expected data, set ReadTimeout long enough to prevent timeouts under normal conditions. Be sure to add in typical delays due to flow control.

An application that must take action in response to received data might need to read each byte as soon as it arrives at the port. In that case, use the DataReceived event for fast response and set ReceivedBytesThreshold to 1.

Sending Data

When sending data, the application should prevent or minimize timeouts and prevent buffer overflows. An application can perform write operations in a separate thread to avoid blocking the application's thread. This section shows strategies for accomplishing these tasks.

Avoiding Timeouts

The likelihood of a timeout increases when an application transfers large quantities of data, uses a slow bit rate, or uses flow control.

When a write operation times out, any data that was waiting to transmit in that operation is lost. Data queued to the port in subsequent write operations remains in the buffer. Setting WriteTimeout to InfiniteTimeout ensures that write operations never time out, but a write operation could wait forever if the remote computer hangs and doesn't allow the data to transmit. Applications that need to send large blocks of data can use multiple write operations to send the data in chunks with generous but finite timeouts. Writing data in smaller chunks also helps prevent buffer overflows.

Sending without Blocking the Application

Using the standard Windows drivers, a write operation to a physical COM port will block until the UART has transmitted all of the data or a timeout has occurred. While the data is transmitting or waiting to transmit, the application's thread can't perform other operations, including responding to user input. Vendor-provided drivers might behave differently but are still likely to block on write operations.

Write operations that complete quickly might have no noticeable or significant effect on performance. Writing large blocks of data, especially at slow bit rates, or writes that experience long delays due to flow control can hang an application for more time than desired.

To enable doing other things while a write operation is in progress, an application can perform the write operation in a separate thread. If needed, the application can assign a routine to be called after the data has transmitted.

To perform write operations in a separate thread, define a delegate for the routine that writes to the port. The example delegate below includes a parameter that holds a string to write to the port.

VB
```
' Define a delegate class to handle writes to the port.

Friend Delegate Function WriteToComPortDelegate(ByVal textToWrite As String) _
                As Boolean

' Create an instance of the delegate.

Friend WriteToComPortDelegate1 _
        As New WriteToComPortDelegate(AddressOf WriteToComPort)
```

VC#
```
// Define a delegate class to handle writes to the port.

internal delegate Boolean WriteToComPortDelegate(string textToWrite);

// Create an instance of the delegate.

internal static WriteToComPortDelegate WriteToComPortDelegate1 =
                new WriteToComPortDelegate(WriteToComPort);
```

When the application has data to send, call the delegate's BeginInvoke method, passing the delegate's parameter (textToWrite), the address of an AsyncCallback routine to execute when the write operation completes (WriteCompleted), and a parameter that can be either Nothing/null or a value to pass to the callback routine (msg):

VB
```
Dim ar As IAsyncResult
Dim msg As String
Dim textToWrite As String

' Text to write to the port.

textToWrite = "hello, world"

' A value to pass to the callback routine (optional).

msg = DateTime.Now.ToString

' Call a routine to write to the COM port.

ar = WriteToComPortDelegate1.BeginInvoke (textToWrite, _
                New AsyncCallback(AddressOf WriteCompleted), msg)
```

VC#

```
IAsyncResult ar;
String msg;
String textToWrite;

// Text to write to the port.

textToWrite = "hello, world";

// A value to pass to the callback routine (optional).

msg = DateTime.Now.ToString();

// Call a routine to write to the COM port.

ar = WriteToComPortDelegate1.BeginInvoke
        (textToWrite, new AsyncCallback(WriteCompleted), msg);
```

The BeginInvoke method calls WriteToComPort, which is the routine named in the declaration of WriteToComPortDelegate1:

VB

```
Friend Function WriteToComPort(ByVal textToWrite As String) As Boolean

    Dim success As Boolean

    If myComPort.IsOpen Then

        ' Write the passed String to the port.

        myComPort.Write(textToWrite)
        success = True
    End If

Return success

End Sub
```

VC#
```
internal static Boolean WriteToComPort( string textToWrite )
{
    Boolean success;

    if (myComPort.IsOpen)
    {
        // Write the passed String to the port.

        myComPort.Write(textToWrite);
        success = true;
    }
    return success;
}
```

After calling BeginInvoke, the application's thread is free to do other things while the write operation is in progress. When the write operation completes, the callback routine executes. The callback routine can perform any actions required after the write operation completes.

VB
```
Friend Sub WriteCompleted(ByVal ar As IAsyncResult)

    Dim msg As String
    Dim deleg As WriteToComPortDelegate
    Dim success As Boolean

    ' To obtain the msg value passed to the BeginInvoke method,
    ' cast BeginInvoke's IAsyncResult value to an AsyncResult object
    ' and get the object's AsyncDelegate property.

    deleg = DirectCast(DirectCast(ar, AsyncResult).AsyncDelegate, _
        WriteToComPortDelegate)

    ' The msg value is in the AsyncState property.

    msg =  = DirectCast(ar.AsyncState, String)

    ' The EndInvoke method returns the value returned by WriteToComPort.

    success = WriteToComPortDelegate1.EndInvoke(ar)

    Console.WriteLine("Write operation began: " & msg)
    Console.WriteLine("Write operation succeeded: " & success)
End Sub
```

VC#
```
internal static void WriteCompleted( IAsyncResult ar )
{
    String msg;
    WriteToComPortDelegate deleg;
    Boolean success;

    // To obtain the msg value passed to the BeginInvoke method,
    // cast BeginInvoke's IAsyncResult value to an AsyncResult object
    // and get the object's AsyncDelegate property.

    deleg = (WriteToComPortDelegate)((AsyncResult)ar).AsyncDelegate;

    // The msg value is in the AsyncState property.

    msg = (String)ar.AsyncState;

    // The EndInvoke method returns the value returned by WriteToComPort.

    success = WriteToComPortDelegate1.EndInvoke(ar);
    Console.WriteLine("Write operation began: " + msg);
    Console.WriteLine("Write operation succeeded: " + success);
}
```

An application can call the delegate multiple times, queuing more data in the transmit buffer with each call.

Preventing Buffer Overflows

The WriteBufferSize property sets the size of the transmit buffer with a minimum size of 2048 bytes. To enable writing more bytes at once to a port for transmitting, increase the size of the transmit buffer.

The safest approach is to check before each write to a port to verify that the transmit buffer has room for the data:

VB

```
Dim stringToWrite as String

stringToWrite = "hello, world"

' Find out if the transmit buffer has room for the new data.

If ((myComPort.WriteBufferSize - myComPort.BytesToWrite) > _
                stringToWrite.Length) Then

                myComPort.Write(stringToWrite)
Else
                Console.WriteLine("Not enough room in transmit buffer.")
End If
```

VC#

```
String stringToWrite;

stringToWrite = "hello, world";

// Find out if the transmit buffer has room for the new data.

if ((myComPort.WriteBufferSize - myComPort.BytesToWrite) > stringToWrite.Length)
{
        myComPort.Write(stringToWrite);
}
else
{
        Console.WriteLine("Not enough room in transmit buffer.");
}
```

Even writing a small amount of data can overflow the write buffer if previous write operations have filled the buffer. A loop or timer routine can check for room in the buffer at intervals and write the data when the buffer has room.

Ensuring Efficient Transfers

As with read operations, each write operation adds overhead. To keep from hanging the application, perform the write operations in a separate thread as described above. Write large amounts of data in one operation or a series of large blocks. Set WriteTimeout long enough to prevent timeouts under normal conditions, including delays due to flow control.

Flow Control

Chapter 2 introduced flow control. The SerialPort class's Handshake property offers four flow-control options: None, Request to Send, Xon/Xoff, and Request to Send with Xon/Xoff.

Selecting a Method

With Request to Send flow control selected, the COM-port driver sets RTS false when the InBufferCount property equals (ReadBufferSize - 1024) bytes or greater. The remote computer should then stop sending data until RTS is true. In the other direction, the driver doesn't send data when CTS is false.

With Xon/Xoff flow control selected, the driver sends an Xoff code when the InBufferCount property reaches (ReadBufferSize - 1024) bytes. The remote computer should then stop sending data until receiving an Xon code. In the other direction, after receiving an Xoff code, the driver doesn't send data until receiving an Xon code.

Request to Send with Xon/Xoff flow control uses both flow-control protocols. If CTS is false indicating that the remote computer can't receive data, the driver won't send an Xoff.

After the local computer asserts RTS or sends an Xoff code, the remote computer can send 1024 bytes before the buffer is full. A computer sending data to such a port should check the flow-control input at least once every 1024 bytes. For example, assume the SerialPort object has 1030 bytes free in its receive buffer and is too busy to process received data immediately. If the sending computer checks the flow-control input, sees that it's OK to send data, and sends 1031 bytes without re-checking, the receive buffer could overflow.

Embedded systems are likely to have much smaller buffers. A computer sending data to embedded systems should check the flow-control input often, ideally before sending each byte.

Monitoring and Controlling the Signals

The SerialPort class also provides properties for monitoring and controlling the flow-control lines explicitly and an event to announce changes at flow-control inputs. An application might need to monitor or control flow-control signals for non-standard uses such as controlling a synchronous serial interface. An application can also send and detect software flow-control codes without relying on the SerialPort object to do so.

Table 10-1: The SerialPort class includes properties for reading and writing to flow-control signals.

Property	Use
BreakState	Gets or sets the state of the TX output. If set to True, no data can transmit on TX.
CDHolding	Gets the state of the CD input.
CtsHolding	Gets the state of the CTS input.
DsrHolding	Gets the state of the DSR input.
DtrEnable	Gets or sets the state the of the DTR output.
RtsEnable	Gets or sets the state the of the RTS output.

Table 10-1 shows the signals that applications can monitor and control. An application can control the RTS and DTR outputs and can read the CD, CTS, and DSR inputs. A True state corresponds to a positive RS-232 voltage, and a False state corresponds to a negative RS-232 voltage. As Chapter 16 explains, the drivers for some USB virtual COM ports don't support reading CTS explicitly, and applications that access these ports can't use the value of CTSHolding.

The BreakState property enables controlling the TX line. Setting the property True brings the RS-232 TX line positive. When TX is True, writing data to the port fails and raises an InvalidOperationException. Setting the property False brings TX negative and enables transmitting data on the line.

The PinChanged event can notify an application when a change has occurred on the CTS, DSR, or CD input, when RI has changed from False to True, and when RX has entered the Break state (but not when RX exits the Break state). RX is in the Break state when the line's state has equaled logic 0 for longer than the time required to transmit one word at the current bit rate.

The code to use the PinChanged event is similar to the code for the DataReceived event earlier in this chapter. The application assigns a routine to execute when a PinChanged event occurs on an open port:

VB
```
' Define a delegate class to handle PinChanged events.

Friend Delegate Sub SerialPinChangedEventHandlerDelegate _
          (ByVal sender As Object, ByVal e As SerialPinChangedEventArgs)

' Create an instance of the delegate.

Private SerialPinChangedEventHandler1 _
          As New SerialPinChangedEventHandler(AddressOf PinChanged)
```

VC#
```
//  Define a delegate class to handle PinChanged events.

internal delegate void SerialPinChangedEventHandlerDelegate
          (object sender, SerialPinChangedEventArgs e);

//  Create an instance of the delegate.

private SerialPinChangedEventHandler SerialPinChangedEventHandler1;

SerialPinChangedEventHandler1 = new SerialPinChangedEventHandler(PinChanged);
```

The application assigns the delegate as the handler routine that will execute when a PinChanged event occurs on an open COM port:

VB
```
AddHandler myComPort.PinChanged, SerialPinChangedEventHandler1
```

VC#
```
myComPort.PinChanged += SerialPinChangedEventHandler1;
```

The routine can check to find out which pin has changed state and take action as needed. This example just displays information about the event:

VB

```
Friend Sub PinChanged(ByVal sender As Object, ByVal e As SerialPinChangedEventArgs)

    Dim SerialPinChange1 As SerialPinChange
    Dim signalState As Boolean

    SerialPinChange1 = e.EventType

        Select Case SerialPinChange1

            Case SerialPinChange.Break

                Console.WriteLine("Break is set")

            Case SerialPinChange.CDChanged

                signalState = myComPort.CDHolding
                Console.WriteLine("CD = " & signalState)

            Case SerialPinChange.CtsChanged

                signalState = myComPort.CtsHolding
                Console.WriteLine("CTS = " & signalState)

            Case SerialPinChange.DsrChanged

                signalState = myComPort.DsrHolding
                Console.WriteLine("DSR = " & signalState)

            Case SerialPinChange.Ring

                Console.WriteLine("Ring detected")

        End Select

End Sub
```

VC#
```
internal void PinChanged( object sender, SerialPinChangedEventArgs e )
{
        SerialPinChange SerialPinChange1 = 0;
        bool signalState = false;

        SerialPinChange1 = e.EventType;
        switch ( SerialPinChange1 )
        {
            case SerialPinChange.Break:

                Console.WriteLine( "Break is set" );
                break;

            case SerialPinChange.CDChanged:

                signalState = myComPort.CDHolding;
                Console.WriteLine( "CD = " + signalState );
                break;

            case SerialPinChange.CtsChanged:

                signalState = myComPort.CtsHolding;
                Console.WriteLine( "CTS = " + signalState );
                break;

            case SerialPinChange.DsrChanged:

                signalState = myComPort.DsrHolding;
                Console.WriteLine( "DSR = " + signalState );
                break;

            case SerialPinChange.Ring:

                Console.WriteLine( "Ring detected" );
                break;
        }
}
```

Handling Errors

Some errors in COM-port communications raise exceptions while others raise the SerialPort object's ErrorReceived event. Application code can also use checksums to detect errors in transmitted data.

Exceptions

This chapter and Chapter 9 have discussed some of the COM-port events and conditions that can raise exceptions. In general, application code can minimize the likelihood of exceptions by checking for valid ports and port states before accessing a port. When needed, a Catch block can display messages to help the user fix the problem by selecting another port, attaching a port, or taking other action.

The ErrorReceived Event

Several error conditions can raise SerialPort's ErrorReceived event. The ErrorReceived code below is similar to the code for the PinChanged and DataReceived events in this chapter.

To use the ErrorReceived event, the application must assign a routine that will execute when an ErrorReceived event occurs on an open port:

VB

```
' Define a delegate class to handle ErrorReceived events.

Friend Delegate Sub SerialErrorReceivedEventHandlerDelegate _
    (ByVal sender As Object, ByVal e As SerialErrorReceivedEventArgs)

' Create an instance of the delegate.

Private SerialErrorReceivedEventHandler1 _
    As New SerialErrorReceivedEventHandler(AddressOf ErrorReceived)
```

VC#

```
// Define a delegate class to handle ErrorReceived events.

internal delegate void SerialErrorReceivedEventHandlerDelegate(object sender,
            SerialErrorReceivedEventArgs e);

// Create an instance of the delegate.

private SerialErrorReceivedEventHandler SerialErrorReceivedEventHandler1;

SerialErrorReceivedEventHandler1 =
        new SerialErrorReceivedEventHandler(ErrorReceived);
```

The application can assign the delegate as the handler routine that will execute when an ErrorReceived event occurs on the port:

VB

```
AddHandler myComPort.ErrorReceived, SerialErrorReceivedEventHandler1
```

VC#

```
myComPort.ErrorReceived += SerialErrorReceivedEventHandler1;
```

The ErrorReceived routine can check to find out which error has occurred and can take action as needed. This example just displays a message on error:

VB
```
Private Sub ErrorReceived _
    (ByVal sender As Object, ByVal e As SerialErrorReceivedEventArgs)

    Dim SerialErrorReceived1 As SerialError

    SerialErrorReceived1 = e.EventType

    Select Case SerialErrorReceived1

        Case SerialError.Frame
            Console.WriteLine("Framing error.")

        Case SerialError.Overrun
            Console.WriteLine("Character buffer overrun.")

        Case SerialError.RXOver
            Console.WriteLine("Input buffer overflow.")

        Case SerialError.RXParity
            Console.WriteLine("Parity error.")

        Case SerialError.TXFull
            Console.WriteLine("Output buffer full.")
    End Select
End Sub
```

VC#
```
private void ErrorReceived(object sender, SerialErrorReceivedEventArgs e)
{
    SerialError SerialErrorReceived1;

    SerialErrorReceived1 = e.EventType;

    switch (SerialErrorReceived1)
    {
        case SerialError.Frame:
            Console.WriteLine("Framing error.");
            break;

        case SerialError.Overrun:
            Console.WriteLine("Character buffer overrun.");
            break;

        case SerialError.RXOver:
            Console.WriteLine("Input buffer overflow.");
            break;

        case SerialError.RXParity:
            Console.WriteLine("Parity error.");
            break;

        case SerialError.TXFull:
            Console.WriteLine("Output buffer full.");
            break;
    }
}
```

The Frame, Overrun, and RXParity errors originate in the UART's Line Status register. The RXOver and TXFull errors report the status of the driver's software buffers.

With a well-structured application and robust hardware design, all of these errors should be rare. Very frequent Framing and Parity errors typically indicate port parameters that don't match on the sending and receiving computers. Occasional Framing or Parity errors can indicate a noisy line that requires a different cable or another hardware fix. Increasing ReadBufferSize can eliminate RXOver errors. To eliminate TXFull errors, ensure that the buffer has room before writing data to the port. Increasing WriteBufferSize can also eliminate TXFull errors and can make the application code more efficient by enabling writing larger blocks of data to the port.

Overrun errors indicate that UART's buffer filled and overflowed because the driver was unable to retrieve data from a full buffer before new data arrived. A slower bit rate can eliminate this error.

Verifying Received Data

As Chapter 2 explained, applications can use hash values and other checksums to verify that received data hasn't been corrupted while transmitting. The System.Security.Cryptography namespace provides classes that implement hash values. To implement error checking with embedded systems that don't support hash values, applications can use simpler checksum algorithms or parity bits.

Structuring an Application

An application that accesses COM ports might define a class for managing COM-port communications, events for passing notifications and data, and combo boxes for setting port parameters. This section shows how to use these elements in applications.

Defining a ComPorts Class

An application can define a class to hold the code that configures and accesses COM ports:

VB

```
Public Class ComPorts

    ' Place class code here.

End Class
```

VC#

```
public class ComPorts
{
    // Place class code here.
}
```

Benefits of using a separate class can include code that is more portable, more readable, and easier to debug. A ComPorts class is likely to have both shared and private members, The shared members can handle tasks that aren't specific to a single port, such as finding and maintaining a record of the system's ports. Private members can handle tasks that are specific to a port, such as opening

and closing the port, sending and receiving data, and storing port parameters and other information about the port.

Applications that need to open multiple ports at the same time can create multiple instances of the class. Each instance of the class can use the common shared members while also maintaining private fields, properties, methods, and events that apply to a specific port.

Chapter 9 showed how to use the GetPortNames method to obtain an array of all of the names of the COM ports in a system, A ComPorts class can implement a nameArray variable as a shared member:

VB
```
Friend Shared nameArray() As String

nameArray = SerialPort.GetPortNames
```

VC#
```
internal static string[] nameArray;

nameArray = SerialPort.GetPortNames();
```

The class can enable accessing a specific COM port by defining a private variable that holds a SerialPort object and a property with Friend scope (internal scope in C#) that provides methods for setting and getting the object:

VB
```
Private m_SelectedPort As New SerialPort

Friend Property SelectedPort() As SerialPort
    Get
        Return m_SelectedPort
    End Get
    Set(ByVal value As SerialPort)
        m_SelectedPort = value
    End Set
  End Property
```

VC#
```
private SerialPort m_SelectedPort = new SerialPort();

internal SerialPort SelectedPort
{
        get
        {
                return m_SelectedPort;
        }
        set
        {
                m_SelectedPort = value;
        }
}
```

To access a specific COM port, an application creates an instance of the ComPorts class:

VB
```
Friend UserPort1 As ComPorts
UserPort1 = New ComPorts
```

VC#
```
internal ComPorts UserPort1;
UserPort1 = new ComPorts();
```

The application can select a port name from the array retrieved with GetPortNames and set SelectedPort's PortName property to the selected name. The application can then set port parameters and open, read, write to, and close the port. The application can call routines in the ComPorts class to set port parameters and open, read, write to, and close the port as needed.

VB
```
Dim comPortIndex as Integer
Dim dataToWrite as String = "hello"
Dim newReceivedData As String

' The index of the selected COM port in the name array.

comPortIndex = 1

UserPort1.SelectedPort.PortName = nameArray(comPortIndex)

UserPort1.SelectedPort.BaudRate = 115200
UserPort1.SelectedPort.Open()
newReceivedData = UserPort1.SelectedPort.ReadExisting
UserPort1.SelectedPort.Write(dataToWrite)
UserPort1.SelectedPort.Close()
```

VC#
```
int comPortIndex;
String dataToWrite = "hello";
String newReceivedData;

// The index of the selected COM port in the name array.

comPortIndex = 3;

UserPort1.SelectedPort.PortName = ComPorts.myPortNames[comPortIndex];

UserPort1.SelectedPort.BaudRate = 115200;
UserPort1.SelectedPort.Open();
newReceivedData = UserPort1.SelectedPort.ReadExisting();
UserPort1.SelectedPort.Write(dataToWrite);
UserPort1.SelectedPort.Close();
```

Setting Parameters with Combo Boxes

The user interface in COM-port applications often provides a way for users to select a COM port and port parameters using combo boxes. The combo boxes can reside on the application's main form, or to minimize clutter, the main form can have a button the user clicks to open a dialog box that contains the combo boxes (Figure 10-1). The _Load event for the dialog box that displays the combo boxes can call a routine that initializes the combo boxes.

To initialize a combo box with the names of the system's ports, use the DataSource property to populate the combo box with the port names retrieved with GetPortNames:

VB
```
cmbPort.DataSource = SerialPort.GetPortNames
```

VC#
```
cmbPort.DataSource = SerialPort.GetPortNames;
```

An application might need to change the contents of this combo box at times to reflect the attachment or removal of virtual COM ports. For example, an application might update the contents every time a user displays the form that contains the combo box. To update the contents, set the DataSource property again.

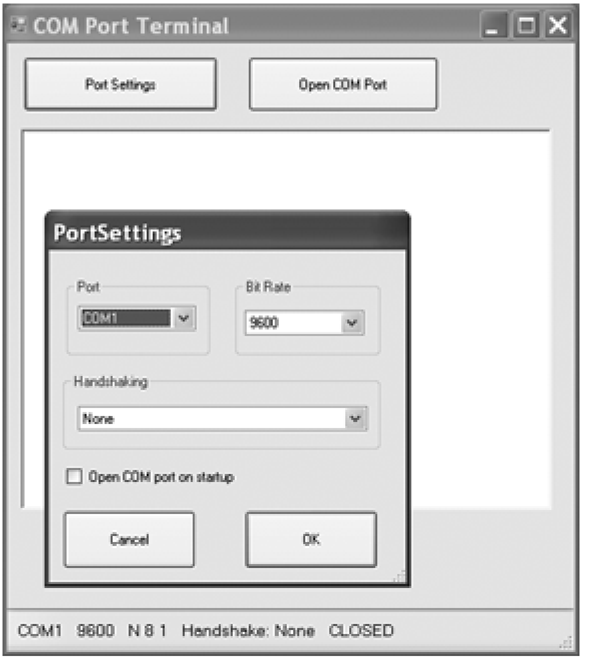


Figure 10-1: In this application, clicking the Port Settings button brings up a window that enables selecting a port and setting port parameters.

To initialize a combo box with available bit rates, create an array of the bit rates and set the DataSource property to the array:

VB

```
Dim bitRates(10) As Integer

 'Bit rates to select from.

bitRates(0) = 300
bitRates(1) = 600
bitRates(2) = 1200
bitRates(3) = 2400
bitRates(4) = 9600
bitRates(5) = 14400
bitRates(6) = 19200
bitRates(7) = 38400
bitRates(8) = 57600
bitRates(9) = 115200
bitRates(10) = 128000
```

```
cmbBitRate.DataSource = bitRates
```

VC#

```
private int[] bitRates = new int[ 11 ];

bitRates[ 0 ] = 300;
bitRates[ 1 ] = 600;
bitRates[ 2 ] = 1200;
bitRates[ 3 ] = 2400;
bitRates[ 4 ] = 9600;
bitRates[ 5 ] = 14400;
bitRates[ 6 ] = 19200;
bitRates[ 7 ] = 38400;
bitRates[ 8 ] = 57600;
bitRates[ 9 ] = 115200;
bitRates[ 10 ] = 128000;

cmbBitRate.DataSource = bitRates;
```

The combo box displays the bit rates as text.

To initialize a combo box with flow-control methods, add each item to the combo box:

VB

```
cmbHandshaking.Items.Add(Handshake.None)
cmbHandshaking.Items.Add(Handshake.XOnXOff)
cmbHandshaking.Items.Add(Handshake.RequestToSend)
cmbHandshaking.Items.Add(Handshake.RequestToSendXOnXOff)
```

VC#

```
cmbHandshaking.Items.Add( Handshake.None );
cmbHandshaking.Items.Add( Handshake.XOnXOff );
cmbHandshaking.Items.Add( Handshake.RequestToSend );
cmbHandshaking.Items.Add( Handshake.RequestToSendXOnXOff );
```

The combo box displays the ToString property of each Handshake object.

In similar ways, you can add combo boxes for setting the number of data bits, parity, and Stop bits as needed.

A combo box's SelectedItem property contains the selected item expressed as text, which code can convert to other object types as needed:

VB
```
' Get the selected bit rate as an Integer.

Dim myBitRate As Integer = _
        CInt(cmbBitRate.SelectedItem)

' Get the selected flow-control method as a Handshake object.

Dim myHandshake As Handshake = _
        DirectCast(cmbHandshaking.SelectedItem, Handshake)

Console.WriteLine(myBitRate)
Console.WriteLine(myHandshake.ToString)
```

VC#
```
// Get the selected bit rate as an Integer.

int myBitRate = (int)(cmbBitRate.SelectedItem);

// Get the selected flow-control method as a Handshake object.

Handshake myHandshake = (Handshake)(cmbHandshaking.SelectedItem);
Console.WriteLine(myBitRate);
Console.WriteLine(myHandshake.ToString());
```

Chapter 9 showed how code can select an item from a combo box by setting its SelectedItem property to match a retrieved user setting. If a statement attempts to set SelectedItem to a value that doesn't exist in the combo box, the combo box ignores the statement and the combo box's SelectedIndex property remains unchanged.

Defining Application-specific Events

Many applications need to pass data to other modules or notify other modules that an event has occurred. For example, a routine that executes on receiving COM-port data might need to pass the received data to the application's form for displaying.

The AccessFormMarshal routine in this chapter showed how a DataReceived routine can pass data to a form in another thread. In the example, the routine explicitly named the instance of the form and the AccessFormMarshal routine.

Events provide a way to pass data and event notifications between modules while requiring only loose coupling between modules. The module that raises the event doesn't have to know anything about a module that handles the event. The module that raises the event just makes the event and its parameters available to any module that wants to receive data or take other action when the event is raised. On being notified of an event, a module can run a routine that accepts passed data and performs any other actions as needed.

For example, a ComPorts class can declare an event that passes text to a form module:

VB

```
Friend Shared Event UserInterfaceData (ByVal textToAdd As String)
```

VC#

```
internal delegate void UserInterfaceDataEventHandler(string textToAdd);

internal static event UserInterfaceDataEventHandler UserInterfaceData;
```

A routine that reads data from the port can raise the event:

VB

```
Dim newReceivedData As String

' Get data from the COM port.

newReceivedData = selectedPort.ReadExisting

' Raise the event to make the data available to other modules.

RaiseEvent UserInterfaceData(newReceivedData)
```

VC#

```
string newReceivedData = null;

// Get data from the COM port.

newReceivedData = SelectedPort.ReadExisting();

// Raise the event to make the data available to other modules.

if (null != UserInterfaceData)
{
    UserInterfaceData(newReceivedData);
}
```

To receive the data, the form's module specifies a routine to execute when the event occurs. The routine must have the same type and number of parameters

as the event. This example names the AccessFormMarshal routine presented earlier in this chapter:

VB
```
AddHandler ComPorts.UserInterfaceData, AddressOf AccessFormMarshal
```

VC#
```
ComPorts.UserInterfaceData +=
        new ComPorts.UserInterfaceDataEventHandler(AccessFormMarshal);
```

When a routine raises the UserInterfaceData event, the AccessFormMarshal routine executes on the form. The AccessFormMarshal routine marshals the textToAdd string to the form for displaying.

In a similar way, other routines can use events to pass data or notify other modules of events. In an application that uses a separate dialog box with combo boxes for setting port parameters, clicking the form's OK button can raise an event that passes the selected port parameters to the main form.

In the Form that contains the combo boxes, declare the event:

VB
```
Friend Shared Event UserInterfacePortSettings(ByVal selectedPort As String, _
        ByVal selectedBitRate As Integer, ByVal selectedHandshake As Handshake)
```

VC#
```
internal delegate void UserInterfacePortSettingsEventHandler
        (string selectedPort, int selectedBitRate, Handshake selectedHandshake);

internal static event UserInterfacePortSettingsEventHandler UserInterfacePortSettings;
```

In the Click event for the form's OK button, pass parameters from the combo boxes. In the example below, the combo boxes are cmbPort, which contains port names, cmbBitRate, which contains bit rates, and cmbHandshaking, which contains handshaking options:

VB
```
RaiseEvent UserInterfacePortSettings(cmbPort.SelectedItem.ToString, _
        CInt(cmbBitRate.SelectedItem), -
        DirectCast(cmbHandshaking.SelectedItem, Handshake))
```

VC#
```
 if ( null != UserInterfacePortSettings )
{
        UserInterfacePortSettings( cmbPort.SelectedItem.ToString(),
                System.Convert.ToInt32( cmbBitRate.SelectedItem ),
                ( ( Handshake )( cmbHandshaking.SelectedItem ) ) );
}
```

In the application's main form, specify a routine that will execute when the event is raised. PortSettingsDialog is the form containing the combo boxes, and SetPortParameters is the routine to execute:

VB

```
AddHandler PortSettingsDialog.UserInterfacePortSettings, AddressOf SetPortParameters
```

VC#

```
PortSettingsDialog.UserInterfacePortSettings += new
    PortSettingsDialog.UserInterfacePortSettingsEventHandler(SetPortParameters);
```

The SetPortParameters routine does whatever is needed with the passed parameters. For example, the routine can set the passed parameters on a SerialPort object:

VB

```
Private Sub SetPortParameters(ByVal userPort As String, ByVal userBitRate As Integer, _
    ByVal userHandshake As Handshake)

    UserPort1.SelectedPort.PortName = userPort
    UserPort1.SelectedPort.BaudRate = userBitRate
    UserPort1.SelectedPort.Handshake = userHandshake

End Sub
```

VC#

```
private void SetPortParameters
    (string userPort, int userBitRate, Handshake userHandshake)
{
    UserPort1.SelectedPort.PortName = userPort;
    UserPort1.SelectedPort.BaudRate = userBitRate;
    UserPort1.SelectedPort.Handshake = userHandshake;
}
```

This page intentionally left blank

11

Ports for Embedded Systems

Many serial-port communications are between embedded systems or between embedded systems and PCs. This chapter shows how to program serial communications for an embedded system's microcontroller. The code examples are for Microchip Technology's PIC18F4520, with each example provided for both microEngineering Labs, Inc.'s PICBASIC PRO compiler and Microchip's MPLAB C18 C compiler. The concepts and much of the code are portable to other chip architectures and compilers.

A Microcontroller Serial Port

Microchip's PIC microcontrollers are inexpensive, widely available, and include variants to suit a range of applications. Many of the chips have hardware support for asynchronous serial communications.

Other microcontrollers support serial ports as well. The details vary, but support typically consists of a hardware UART or USART with a series of registers to configure the port, monitor events, and store received data and data to send.

Compilers often provide library functions that simplify accessing the port hardware.

About the PIC18F4520

The PIC18F4520 has an 8-bit data bus, 32 kilobytes (KB) of flash memory for program storage, and 1.5 KB of RAM and 256 bytes of EEPROM for data storage. The power supply can range from +4.2V to +5.5V. The PIC18LF4520 variant can use a power supply as low as +2V.

For serial communications, the chip contains an enhanced USART (EUSART) that supports asynchronous and synchronous serial communications. Other on-chip functions include four timers, a 10-bit analog-to-digital converter with up to 13 channels, an analog comparator, and two modules that can perform capture, compare, and pulse-width-modulation (PWM) functions. The 40/44-pin versions also have an 8-bit parallel port. Most of the chips' I/O pins have assigned functions, such as serial-port input and output, but the pins can have other uses if firmware isn't using the assigned functions.

Other chips in the PIC18F4520 series have different combinations of functions and package options. The data sheet for the series has complete documentation for the chips.

The Enhanced UART

The PIC18F4520's EUSART supports using either an asynchronous or synchronous interface but not both at the same time. However, the PIC18F4520 has an additional synchronous serial port module that supports synchronous communications. The enhanced features of the EUSART include automatic bit-rate detecting and a 16-bit register for finer control of bit rates. Other PIC microcontrollers have similar USARTs, though not every variant has the enhanced features of the EUSART.

The PIC18F4520 dedicates two port bits for the asynchronous port. Port C, bit 6 is the data output (TX), and Port C, bit 7 is the data input (RX). Hardware flow control can use any spare port pins. The pins can interface directly to most TTL and CMOS logic chips, including the TTL/CMOS interfaces on many RS-232 and RS-485 converter chips.

Registers

Firmware accesses the PIC18F4520's asynchronous serial port via a series of registers. When a register bit enables and disables a feature, a value of 1 enables the feature and zero disables the feature.

Configuring and Accessing the Port

Registers hold received data, data waiting to transmit, and status and control information for the port. Figure 11-1 shows how data travels between the port pins and the CPU.

EUSART Receive Data Register (RCREG)

The EUSART receive data register (RCREG) holds up to two bytes of received data in a first-in, first-out (FIFO) buffer. Firmware reads data from the buffer in the same order as it was received.

EUSART Transmit Data Register (TXREG)

The EUSART transmit data register (TXREG) holds one byte of data to transmit.

Receive Serial Shift Register (RSR)

The receive serial shift register (RSR) stores bits received on the RX pin. After detecting a Stop bit, if RCREG has room, the RSR loads the received byte into RCREG. If RCREG is full, the RSR waits and loads the byte when space becomes available. Firmware can't access the RSR directly.

Transmit Serial Shift Register (TSR)

The transmit serial shift register (TSR) loads data from TXREG and writes the bits to the TX pin. The TSR loads data when TXREG contains a byte to send and the TSR is empty. Firmware can't access the TSR directly.

Transmit Status and Control Register (TXSTA)

The transmit status and control register (TXSTA) contains bits used in configuring and enabling the port. The register also holds the optional ninth bit for transmitting. To enable receiving 8-bit asynchronous data, set TXSTA to 20h or 24h.

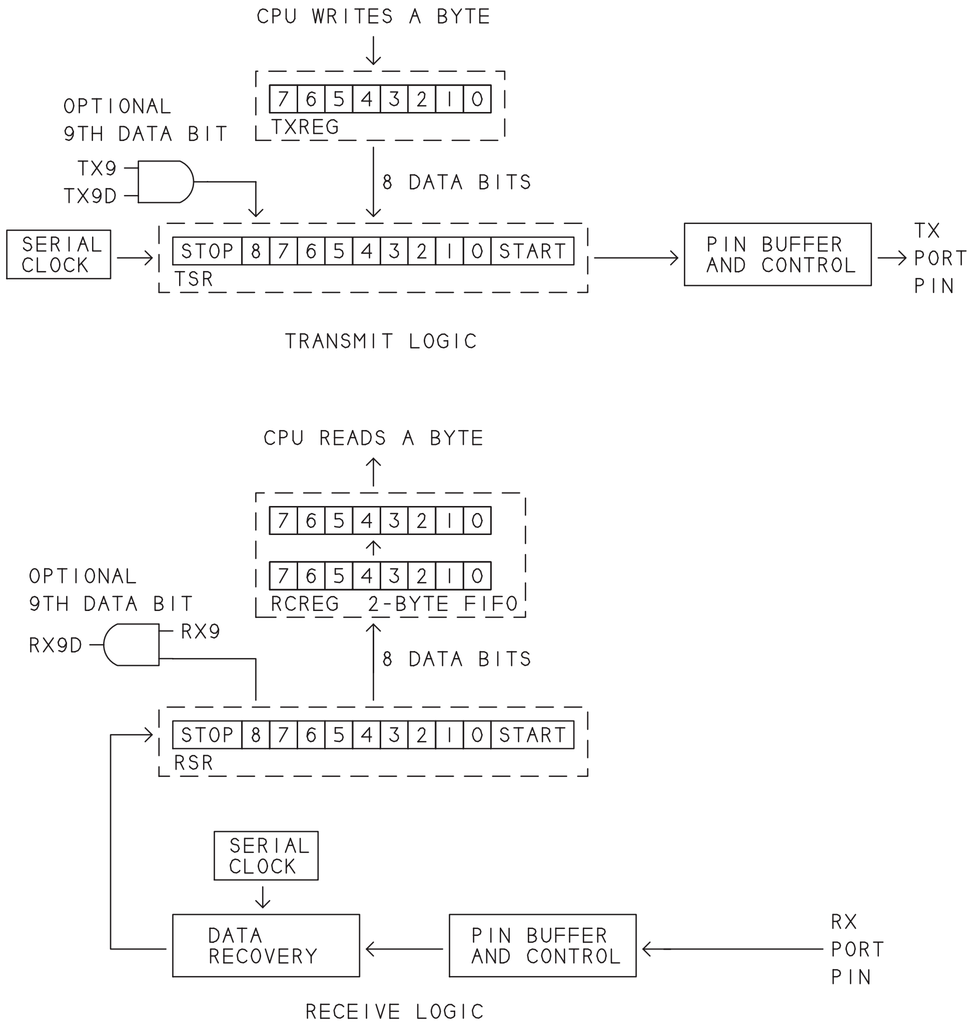


Figure 11-1: The CPU reads and writes to the serial port by accessing registers.

Bit 7 isn't used in asynchronous communications and should maintain the default value.

Bit 6 (TX9) sets the number of bits to transmit to 8 (0) or 9 (1).

Bit 5 (TXEN) enables transmitting.

Bit 4 (SYNC) is set to zero for asynchronous communications.

Bit 3 (SENDB) is set to zero for normal communications and is set to 1 to enable sending a Sync Break character, which is a special character defined by the LIN bus protocol used in automotive applications. The character consists of a Start bit, 12 zeros, and a Stop bit. When SENDB = 1, TXEN = 1, and a value is loaded from TXREG into the TSR, the EUSART ignores the data in the TSR and instead sends a Sync Break character.

Bit 2 (BRGH) selects whether to use the high speed (1) or low speed (0) value for the bit rate specified in the SPBRG and SPBRGH registers as described later in this chapter.

Bit 1 (TRMT) indicates whether the TSR is empty (1) or full (0). This bit doesn't control an interrupt but firmware can read the bit.

Bit 0 (TX9D) holds the ninth bit of data to send when TX9 = 1. Firmware must provide the value. If used, firmware should write a value to TX9D before writing a byte to send to TXREG.

Receive Status and Control Register (RCSTA)

The receive status and control register (RCSTA) enables monitoring and controlling received data. The register also contains the SPEN bit, which enables the serial port by configuring the dedicated port pins for serial data. The RX9D bit holds an optional received ninth data or parity bit. To enable receiving 8-bit asynchronous data, set the register to 90h.

To ensure reading valid values, firmware should read any bits of interest in RCSTA before reading RCREG to obtain a received byte. Reading RCREG causes bits RX9D, OERR, and FERR to update for the next received byte.

Bit 7 (SPEN), enables the serial port by configuring the RX and TX pins as the input and output for USART data.

Bit 6 (RX9) enables receiving 9-bit data.

Bit 5 (SREN) has no function in asynchronous communications and should maintain the default value.

Bit 4 (CREN) enables the receiver.

Bit 3 (ADDEN) enables the address-detect feature used with 9-bit data.

Bit 2 (FERR) is set to 1 on detecting a framing error, which occurs on detecting a logic-low Stop bit at the RX pin.

Bit 1 (OERR) is set to 1 on detecting an overrun error. The error occurs when RCREG holds two unread bytes and the RSR detects the Stop bit of a new received word. The RSR has nowhere to load the new byte, so the byte and any data that follows while OERR is set is lost. Firmware is responsible for reading bytes from RCREG frequently enough to prevent overrun errors. Hardware flow control can help by de-asserting the flow-control output when RCREG is full. To clear OERR, set CREN = 0. Then set CREN = 1 to re-enable receiving data.

Bit 0 (RX9D) holds the ninth bit of received data when RX9 = 1. If the bit is a parity bit, firmware is responsible for reading the bit and detecting parity errors.

Setting the Bit Rate

The serial port's bit rate depends on a variety of register settings and the frequency of the CPU's clock source, or FOSC. The chips support multiple operating modes that can use different clock sources to conserve power when possible.

Baud Rate Control Register (BAUDCON)

The Baud Rate Control Register (BAUDCON) contains bits that relate to enhanced features of the chip's bit-rate generator for serial communications.

Bit 7 (ABDOVF) = 1 when a rollover has occurred while in auto-baud detect mode. Firmware must clear the bit.

Bit 6 (RCIDL) = 1 when the receiver is idle (no receive operation is in progress).

Bit 5 (RXDTP) sets the polarity for received data as inverted (1) or non-inverted (0).

Bit 4 (TXCKP) sets the polarity for the idle state as low (1) or high (0).

Bit 3 (BRG16) determines whether to use an eight (0) or sixteen (1) bit value to set the bit rate. In general, 16 bits enables greater accuracy in selecting rates.

Bit 2 is unimplemented and reads zero.

Bit 1 (WUE) is set to 1 to cause the EUSART to trigger an interrupt on a falling edge at RX. The interrupt can wake the chip from Sleep mode.

Bit 0 (ADBDEN) is set to 1 to cause the chip to detect the bit rate on the next received byte. The received byte must equal 55h for accurate detecting of the rate.

Oscillator Control Register (OSCCON)

In the oscillator control register (OSCCON), the SCS1 and SCS0 bits select the CPU's clock source. In normal, full-power operation, the clock source is the primary clock specified in CONFIG1H. For reduced power consumption, the clock source can be the Timer1 oscillator or the internal oscillator block. When using the internal oscillator block, the IRCF2..IRCF0 bits select the frequency. When using the Timer1 oscillator, the frequency is determined by the clock source connected to the T1OSI and T1OSO pins.

After changing the clock source, firmware that wants to continue using the serial port will likely need to change the values in the SPBRG and SPBRGH registers as described below.

Oscillator Tuning Register (OSCTUNE)

The oscillator tuning register (OSCTUNE) provides additional control of the internal oscillator including selecting a source for the internal 31-kHz oscillator (INTSRC), enabling a 4x frequency multiplier (PLLEN), and fine-tuning the oscillator's frequency (TUN4..TUN0).

Configuration Register 1 High (CONFIG1H)

When OSCCON has configured the CPU to use the primary clock, Configuration register 1 high (CONFIG1H) selects the clock's source. FOSC3..FOSC0 set the source as a timing input on the OSC1 and OSC2 pins (in some cases only OSC1 is used) or as the chip's internal oscillator block.

Depending on the values of the FOSC bits, the primary clock's frequency can be the same as the frequency at OSC1 and OSC2 or four times that value. Setting FOSC3..FOSC0 to 0001_b configures the chip to use an external oscillator for FOSC. A 10-MHz oscillator results in FOSC = 10 MHz. Setting the bits to 0110_b enables using slower external clocks by setting FOSC equal to the external oscillator's frequency multiplied by four. A 10-MHz oscillator results in FOSC = 40 MHz.

The functions of the bits in the configuration registers vary with the chip, so always check the data sheet for the chip you're using.

Baud Rate Generator Registers (SPBRG and SPBRGH)

The baud rate generator registers (SPBRG and optionally SPBRGH) contain a value used in setting the bit rate. If the BAUDCON register's BRG16 = 0, the value is 8 bits and SPBRG contains the entire value. If BRG16 = 1, the value is 16 bits, SPBRG contains the low byte, and SPBRGH contains the high byte.

Steps in Selecting a Bit Rate

To set the register values for a desired bit rate, do the following:

1. Obtain the value of FOSC. Begin by determining the clock source from the OSCCON register (SCS1..SCS0). If using the primary clock, determine FOSC from the frequency of the clock source and the value of FOSC3..FOSC0 in the CONFIG1H register. If using the internal oscillator block, determine FOSC from the settings in the OSCCON and OSCTUNE registers. If using the Timer1 oscillator, FOSC is the frequency of the timing source connected to T1OSI and T1OSO.

2. Determine the values to store in the registers. Two ways to do so are by consulting the tables provided in the data sheet and by calculating the values.

For many applications, you can find a match in the tables in the chip's data sheet. For common FOSC values, the tables show four options depending on the values of BRGH in the TXSTA register and BRG16 in the BAUDCON register. Select the value with the smallest error. For example, if FOSC = 20 MHz and you want a bit rate of 115,200 bps, the smallest error (0.94%) is with BRGH = 1, BRG16 = 1, SPBRG = 42 and SPBGRH = 0.

If your bit rate or FOSC value isn't in the tables, you'll need to calculate the best value. Computer assistance is useful. See *www.Lvr.com* for a link to an application that selects values for a desired bit rate and oscillator frequency

To find a value to store in SPBRG (and SPBRGH if used) for a desired bit rate and FOSC, BRGH, and BRG16 settings, use this formula:

```
spbrg = Math.Round((fosc / (multiplier * bitRate)) - 1)
```

spbrg is the 8-bit value in SPBRG or the 16-bit value in SPBRGH and SPBRG. If BRG16 = 0, SPBRG must be 255 or less. If the calculated *spbrg* value is slightly higher than 255, setting SPBRG to 255 might result in a usable bit rate.

fosc is the FOSC value in Hz.

The value of the *multiplier* variable depends on the values in BRG16 and BRGH:

BRGH	BRG16	Multiplier
0	0	64
0	1	16
1	0	16
1	1	4

If *multiplier* = 16 and *spbrg* < 256, firmware can achieve the same bit rate by setting BRGH = 1 and BRG16 = 0 or by setting BRGH = 0 and BRG16 =1. If *multiplier* = 16 and *spbrg* > 255, firmware must set BRGH = 0 and BRG16 = 1.

bitRate is the desired bit rate.

To find the setting that results in the closest bit rate to the desired value, perform the calculation with each of the three multiplier values. This formula calculates the actual bit rate that results from specific values:

```
actualBitRate = Round(fosc / (multiplier * (spbrg + 1)))
```

The *fosc* and *multiplier* variables are set as described above. Depending on the value of BRG16, *spbrg* is the 8-bit value in SPBRG or the 16-bit value in SPBRGH and SPBRG. If BRG16 = 1 and *spbrg* is greater than 255, you need to divide spbrg into its lower and higher bytes:

```
spbrgl = spbrg Mod 256
spbrgh = Int(spbrg / 256)
```

A carefully selected FOSC frequency can result in a calculated bit rate with zero error. However, the hardware oscillator can still introduce error if the oscillator doesn't operate at exactly its rated frequency.

Interrupts

Firmware can use hardware interrupts to cause program code to branch to an interrupt service routine (ISR) when the serial port has received data or when the port's transmit buffer is empty and ready to store new data to send. Firmware can also set priority levels for specific interrupts to service critical interrupts as quickly as possible. For serial communications at fast bit rates, using a high-priority interrupt can be useful in preventing overflows of the receive buffer.

For register bits that enable and disable specific interrupts or groups of interrupts, a value of 1 enables the interrupt and a value of 0 disables the interrupt.

Peripheral Interrupt Request Register 1 (PIR1)

The peripheral interrupt request register 1 (PIR1) contains bits that indicate the status of individual peripheral interrupt sources.

Bit 5 (RCIF) indicates whether RCREG contains at least one byte (1) or is empty (0).

Bit 4 (TXIF) indicates whether TXREG is full (0) or empty (1). The bit is set to 1 when the TXSTA register's TXEN bit is set to 1 and thereafter whenever the TSR loads a byte to transmit from TXREG. The bit is set to zero when firmware writes a byte to TXREG.

Note that RCIF = 1 when its buffer contains data and TXIF = 1 when its buffer is empty. These are the conditions when firmware is likely to need to take action. If the interrupts aren't enabled, firmware can read the bits to learn the states of the buffers.

Reset Control Register (RCON)

The Reset Control Register (RCON) has one bit that can affect serial interrupts.

Bit 7 (IPEN) enables priority levels on interrupts.

Peripheral Interrupt Priority Register 1 (IPR1)

The Peripheral Interrupt Priority Register 1 (IPR1) has two bits that set the interrupt priority level for serial-port communications. These levels take effect if the RCON register's IPEN bit = 1. A value of 1 sets high priority, and zero sets low priority. A high-priority interrupt will interrupt a low-priority interrupt being serviced.

Bit 5 (RCIP) sets the receive interrupt priority.

Bit 4 (TXIP) sets the transmit interrupt priority.

Interrupt Control Register (INTCON)

The interrupt control register (INTCON) has two bits that affect serial-port interrupts. The functions of the bits vary depending on the value of the IPEN bit in the RCON register.

When IPEN = 0:

Bit 7 (GIE) enables or disables all unmasked interrupts.

Bit 6 (PEIE) enables or disables all unmasked peripheral interrupts.

When IPEN = 1:

Bit 7 (GIEH) enables or disables all high-priority interrupts.

Bit 6 (GIEL) enables or disables all low-priority peripheral interrupts.

Peripheral Interrupt Enable Register 1 (PIE1)

The peripheral interrupt enable register 1 (PIE1) enables interrupts for a variety of peripheral sources. Two bits relate to serial ports.

Bit 5 (RCIE) enables the serial-port's receive interrupt. The interrupt occurs when the PIR1 register's RCIF bit = 1.

Bit 4 (TXIE) enables the serial port's transmit interrupt. The interrupt occurs when the PIR1 register's TXIF bit = 1.

Basic Operations

Using a serial port involves the operations of enabling the port, configuring interrupts if used, and transmitting and receiving data.

Enabling the Port

To enable the asynchronous serial port for transmitting and receiving 8-bit data, firmware should perform the following tasks:

1. Select values to configure the chip for the desired bit rate. Set the TXSTA register's BRGH bit and the BAUDCON register's BRG16 bit to the desired values. To set the bit rate, write a value to SPBRG and to SPBRGH if used.
2. In the TXSTA register, set SYNC = 0 to select asynchronous mode.
3. In the RCSTA register, set SPEN = 1 to enable the serial port.

Using Interrupts

To use the transmit and received interrupts without using interrupt priority:

1. In the PIE1 register, set TXIE = 1 and RCIE = 1.
2. In the INTCON register, set GIE = 1 and PIE = 1.

To use the transmit and received interrupts with interrupt priority:

1. In the RCON register, set IPEN = 1.
2. In the PIE1 register, set TXIE = 1 and RCIE = 1.
3. In the IPR1 register, set the interrupt priority in RCIP and TXIP as desired.
4. In the INTCON register, set GIEH and GIEH to enable interrupts as appropriate for the IPR1 settings.

Transmitting

To transmit a byte, firmware performs these tasks:

1. To enable transmitting, in the TXSTA register, set TXEN = 1.
2. If using hardware flow control, check the state of the firmware-defined flow-control input bit and wait as needed for permission to transmit.
3. Write a byte to transmit to TXREG. The byte transfers to the TSR. When TXREG is empty, in the PIR1 register, TXIF = 1. On the TX pin, the TSR transmits a Start bit, the byte of data, and a Stop bit.

Firmware can write another byte to TXEN as soon as TXIF = 1. If using interrupts, firmware will jump to the ISR when TXIF = 1.

Receiving

To receive a byte, firmware performs these tasks:

1. To enable receiving, in the RCSTA register, set CREN = 1.
2. If using hardware flow control, set the firmware-defined flow-control output bit to enable receiving.
3. In the PIR1 register, wait for RCIF = 1, indicating the receive buffer contains as least one byte. If using interrupts, firmware will jump to the ISR when RCIF = 1.
4. In the RCSTA register, check the framing error (FERR) and overrun (OERR) bits.
5. Read the received byte in RCREG. Reading the register clears RCIF. If FERR = 1, firmware will likely want to discard the received byte.

6. If the RCSTA register indicated an error, set CREN = 0 to clear the error. To enable receiving another byte, set CREN = 1.

Accessing a Port

Both the PICBASIC PRO and MPLAB C18 compiler provide support for configuring and reading and writing to serial ports. The PICBASIC PRO compiler defines statements for accessing up to two hardware serial ports and two firmware-only serial ports. The C18 compiler provides a series of library functions for accessing serial ports. For chips that have more than one USART, the C18 compiler enables firmware to specify which USART to access. The compiler also provides functions that implement one or more firmware UARTs using any spare port pins. The examples that follow are for hardware UARTs.

Configuring the Port

To prepare to use the serial port for asynchronous communications, firmware must set the EUSART to asynchronous mode, set the bit rate and number of data bits, enable the port and the receiver, and enable the transmit and receive interrupts if desired.

PBP DEFINE statements can set many port parameters, including the values in the TXSTA and RCSTA registers:

```
DEFINE HSER_RCSTA 90h
DEFINE HSER_TXSTA 20h
```

The HSER_BAUD definition sets the value in the SPBRG register for the specified bit rate using the selected setting of the TXSTA register's BRGH bit:

```
DEFINE HSER_BAUD 2400
```

An alternate way to set the bit rate is to write a value directly to SPBRG:

```
DEFINE HSER_SPBRG 19h
```

If FOSC doesn't equal the default value of 4 MHz, include a DEFINE OSC statement to specify the FOSC frequency in MHz:

```
DEFINE OSC 20
DEFINE HSER_BAUD 2400
```

The example below assumes a 4-MHz oscillator and sets the bit rate to 2400 using a 16-bit value. In the TXSTA register, bit 2 (BRGH) is set to 1 to select using the high-speed value. In the BAUDCON register, bit 3 (BRG16) is set to 1 to enable using a 16-bit value in SPBRG and SPBRGH. The value stored in the registers is 19Fh.

```
DEFINE HSER_TXSTA 24h
DEFINE HSER_SPBRGH 1
DEFINE HSER_SPBRG 9fh
BAUDCON = 8
```

C18 The OpenUSART function sets the port parameters. The statement below disables the transmit and receive interrupts, sets the EUSART to asynchronous mode, configures the port for 8 data bits, enables the receiver, and sets the bit rate to 2400 bps assuming FOSC = 4 MHz. The final parameter is the value for the SPBRG register.

```
BAUDCONbits.BRG16 = 0;

OpenUSART (USART_TX_INT_OFF &
                    USART_RX_INT_OFF &
                    USART_ASYNCH_MODE &
                    USART_EIGHT_BIT &
                    USART_CONT_RX &
                    USART_BRGH_LOW,
                    0x19);
```

This example is identical except FOSC = 20 MHz, which requires a different value in SPBRG:

```
OpenUSART (USART_TX_INT_OFF &
                    USART_RX_INT_OFF &
                    USART_ASYNCH_MODE &
                    USART_EIGHT_BIT &
                    USART_CONT_RX &
                    USART_BRGH_LOW,
                    0x81);
```

Using a 16-bit value to set the baud rate requires setting BRG16 = 1 and storing a value in SPBRGH. This example sets a bit rate of 2400 bps assuming FOSC = 4 MHz:

```
BAUDCONbits.BRG16 = 1;
SPBRGH = 1;

OpenUSART (USART_TX_INT_OFF &
                    USART_RX_INT_OFF &
                    USART_ASYNCH_MODE &
                    USART_EIGHT_BIT &
                    USART_CONT_RX &
                    USART_BRGH_HIGH,
                    0x9f);
```

Sending Data

Firmware can write individual bytes or arrays of bytes to a port. Before writing to a port, firmware should wait for TXIF = 0, indicating that the port isn't busy sending a byte.

PBP The hserout statement waits for TXIF = 0 and writes data to the port.

Each of the hserout statements below write the same value (41h, the code for the character "A") to the port:

```
hserout ["A"]
hserout [$41]

test var byte
test = $41
hserout[test]
```

An hserout statement can also write multiple bytes to a port. This statement writes a string:

```
hserout ["hello, world"]
```

This statement writes the values stored in a byte array:

```
test var byte[2]

test[0] = "O"
test[1] = "K"
hserout[STR test]
```

C18 The putcUSART function writes a byte to the port. The BusyUSART function returns 1 when the port is busy transmitting. To ensure that the port is available, place a while(BusyUSART()) statement before attempting to send data. The WriteUSART function is identical to putcUSART.

Both of the putcUSART statements below write the same value (41h, the code for the character "A") to the port:

```
while(BusyUSART());
putcUSART('A')

while(BusyUSART());
putcUSART(0x41)
```

The putsUSART and putrsUSART functions can write strings to a port. This example uses putrsUSART to write a string, including a null terminator (0), from program memory to the port:

```
putrsUSART((rom char*) "Hello World!" );
```

This example uses putsUSART to write a character array from data memory up to and including the null terminator:

```
unsigned char my_string[3] = {'\0'};

my_string[0] = 'O';
my_string[1] = 'K';
my_string[2] = '\0';

putsUSART((char*)my_string);
```

Receiving Data

Firmware that reads bytes at the serial port can check for errors, store received bytes in an array, and perform other tasks while waiting for data to arrive. A protocol can define a terminating character to indicate the end of a command or string.

Reading a Byte

When receiving data, firmware can read the RCIF bit in the PIR1 register to find out if a byte is available for reading.

PBP An hserin statement can read a byte at the serial port:

```
serial_in		var	byte

if (PIR1.5 = 1) then

	' A byte is available. Read it.

	hserin [serial_in]
endif
```

Instead of checking RCIF before calling hserin, a statement can call hserin with a timeout that jumps to a label if a byte doesn't arrive. Reading RCIF is more efficient because the code doesn't waste time waiting, but a timeout can be useful if the application is expecting data and has no other urgent tasks.

```
serial_in		var	byte
timeout		var	word

timeout = 1000

' Wait for a byte on the serial port.

hserin timeout, give_up, [serial_in]

' A byte was received. Do something with it.

' Jump here if hserin times out.

give_up:
```

C18 The DataRdyUSART function returns 1 when RCIF =1. The getcUSART function reads a byte from the port. The ReadUSART function is identical to getcUSART.

```
unsigned char serial_in;

if DataRdyUSART()
{
	// A byte is available.

	serial_in = getcUSART();
	}
```

Detecting Errors

Firmware can check for framing or overrun errors on the port. The following examples show how to check for these errors. To avoid repetition, the additional code examples in this book don't check for overrun or framing errors, but most working code should include the code below or an equivalent when reading data from a port.

PBP This statement causes the code to automatically clear any overrun errors:

```
DEFINE HSER_CLROERR
```

Code can instead detect and clear overrun errors as they occur:

```
if (RCSTA.1 = 1) then

    ' Overrun error. Clear and set CREN to clear the error and re-enable the receiver.

    RCSTA.4 = 0
    RCSTA.4 = 1

    ' (Perform any other required actions.)
endif
```

To check for framing errors, firmware must read the RCSTA register after the byte has arrived at the port and before firmware reads the byte from RCREG:

```
if (PIR1.5 = 1) then

    ' A byte has arrived.

    if (RCSTA.2 = 1) then

        ' Framing error.
        ' Read RCREG to clear the error but don't use the data.

        hserin [serial_in]

    else
        ' No framing error occurred. The data is valid.

        hserin [serial_in]

        ' Do something with the received byte.
    endif
endif
```

If the firmware uses hserin to read multiple bytes or to wait for a byte, there is no way to detect overrun and framing errors because firmware has no way to read RCSTA after executing hserin but before reading RCREG. For some applications, detecting framing and overrun errors isn't essential.

C18 After determining that a byte is available and before reading the byte, code can check for overrun and framing errors.

```
unsigned char serial_in;

if DataRdyUSART()
{
	// A character is available.

	if (RCSTAbits.OERR == 1)
	{
		// Overrun error.
		// Clear the error and re-enable the receiver.

		RCSTAbits.CREN = 0;
		RCSTAbits.CREN = 1;
	}
	if (RCSTAbits.FERR == 1)
	{
		// Framing error.
		// Read the byte to clear the error but don't use the byte.

		serial_in = getcUSART();
	}
	else
	{
		// A byte was received without error.

		serial_in = getcUSART();

		// Do something with the byte.
	}
}
```

Using a Task Loop

A task loop is an endless loop that can perform serial-communications tasks and any other tasks a device is responsible for. Firmware performs each task in

sequence, starting over on reaching the end of the task list. For example, one task can check for received serial-port data while another task reads and stores sensor data from another port.

PBP
```
serial_in        var   byte
timeout          var   word

loop:
    'Task loop.

    gosub receive_serial_data

    ' Call routines fo perform other tasks.

goto loop

receive_serial_data:

    if (PIR1.5 = 1) then

        ' A byte is available. Read it.

        hserin [serial_in]

        ' Do something with the received byte.

    endif

return
```

C18
```
unsigned char serial_in;

while(1)
{
	// Task loop.
	receive_serial_data();
	// Call routines to perform other tasks.
}

void receive_serial_data(void)
{
	unsigned char serial_in;

	if DataRdyUSART()
	{
		// A byte is available.

		serial_in = getcUSART();

		// Do something with the received byte.
	}
}
```

Reading Multiple Bytes

When expecting multiple bytes, firmware can wait for the bytes to arrive or read and store individual bytes until all have arrived.

PBP An hserin statement can wait to receive a specified number of bytes to store in a byte array. This example waits for 7 bytes and stores each received byte in sequence in received_text:

```
received_text		var	byte[7]

hserin [STR received_text\7]
```

A receiving computer can't guarantee that a remote computer will send the desired number of bytes. Using a timeout will keep the code from hanging forever while waiting for bytes to arrive.

```
received_text		var	byte[7]
hserin 1000, continue, [STR received_text\7]

continue:
```

On a timeout, the STR array contains any bytes that arrived. If you need to know how many bytes were received, initialize the STR array with a value that the remote computer doesn't transmit. For example, if receiving text data, initialize the array with null characters (zeros) before calling herserin:

```
for index = 0 to 9
	received_text[index] = 0
next index
```

Another way to read multiple bytes is to read one byte at a time and store the bytes in an array. A task loop can call the routine below to read a specified number of bytes into an array:

```
index var byte

receive_serial_data:

	received_data var byte[8]
	bytes_to_read var byte

	bytes_to_read = 8

	if (PIR1.5 = 1) then

		' A byte is available to read.

		hserin [received_data[index]]

		index = index + 1

		if index >= bytes_to_read then

			' Do something with the received bytes.

			' Reset the index for the set of received bytes.

			index = 0
		endif
	endif
return
```

C18 The getsUSART function can read multiple characters but requires specifying how many characters to read and blocks until all of the characters arrive:

```
char received_data[8];

getsUSART(received_data, 8);
```

To read a string without blocking, firmware can call the getcUSART function repeatedly to read individual bytes into a char array. A task loop can call the routine below to read a specified number of bytes into an array:

```
byte index;

void receive_serial_data(void)
{
	byte bytes_to_read;
	char received_data[8];

	bytes_to_read = 8;

	if DataRdyUSART()
	{
		// A byte is available. Read it into the array.

		received_data[index] = getcUSART();

		// Increment the position in the array.

		index++;

		if (index == bytes_to_read)
		{
			// Do something with the received bytes.

			// Reset the index for the set of received bytes.

			index = 0;
		}
}
```

Detecting a Terminating Character

A defined terminating character can indicate the end of a command, string, or other block of data. The examples below check for a CR character (0Dh) in received data.

PBP An hserin statement can return on receiving a specified value or receiving the specified number of bytes, whichever comes first. This statement waits to receive 10 bytes or a CR code:

```
hserin [STR received_text\10\$0d]
```

Another option is to store bytes as they arrive and check each byte for the CR code. Reading individual bytes in a task loop eliminates blocking while waiting for the CR.

```
index                   var     byte
received_data           var     byte[80]

receive_serial_data:

        if (PIR1.5 = 1) then

                ' A byte is available to read.

                hserin [received_data[index]]

                if received_data[index] = 13 then

                        ' It's a CR code. Do something with the received data.
                        ' Reset the index.

                        index = 0
                else
                        ' Increment the index, resetting to 0 if the buffer is full.

                        if index < 79 then
                                index = index + 1
                        else
                                index = 0
                        endif
                endif
        endif
return
```

C18 A device that is receiving commands that end with a CR can check for the character on receiving data and take action when received:

```
byte index;
char received_data[80];

void receive_serial_data(void)
{
        if DataRdyUSART()
        {
                // A byte is available.

                received_data[index] = getcUSART();

                if (received_data[index] == 0x0d)
                {
                        // It's a CR code. Do something with the received data.
                        // Reset the index.

                        index = 0;
                }
                else
                {
                        // Increment the index, resetting to 0 if the buffer is full.

                        if (index < 79 )
                                index++;
                        else
                                index = 0;
                }
        }
}
```

Using Interrupts

When data arrives at unpredictable times, firmware can use the receive interrupt to read bytes as they arrive.

With the receive interrupt enabled, the CPU jumps to the interrupt vector (a defined location in program memory) every time a byte arrives at the port. An ISR can read the received byte and perform any other needed actions.

In a similar way, with the transmit interrupt enabled, the CPU jumps to the interrupt vector every time TXREG is newly empty. The ISR can then write

another byte to TXREG for transmitting and perform any other necessary actions.

The PIC18F4520 has two interrupt vectors. Multiple interrupts with the same priority share an interrupt vector. If not using interrupt priority, all interrupts use vector 08h. If using interrupt priority, the high-priority interrupts use vector 08h and the low-priority interrupt uses vector 18h. If more than one interrupt is enabled and uses the same vector, the code in the ISR must read the interrupt flags to determine the source of the interrupt and take appropriate action.

PBP To use the receive interrupt, set the appropriate values in the INTCON and PIE1 registers:

```
' Enable unmasked peripheral interrupts

INTCON = %11000000

' Enable the serial receive interrupt

PIE1 = %00100000
```

The ON INTERRUPT statement specifies what routine to run when an interrupt occurs:

```
ON INTERRUPT GOTO interrupt_service_routine
```

The ISR begins with a label and ends with a RESUME statement. A DISABLE statement before the label holds all interrupt processing except for the current interrupt until the next ENABLE statement.

```
DISABLE
interrupt_service_routine:

    if (PIR1bits.RCIF == 1)
        ' A serial receive interrupt has occurred.
        ' Place code to read and respond to received data here.
    else
        ' Check other flags for enabled interrupts and take action as needed.
    endif
RESUME
ENABLE
```

C18 The OpenUSART function enables the receive interrupt by setting the second parameter to USART_RX_INT_ON:

```
OpenUSART (USART_TX_INT_OFF &
        USART_RX_INT_ON &
        USART_ASYNCH_MODE &
        USART_EIGHT_BIT &
        USART_CONT_RX &
                USART_BRGH_LOW,
        77);
```

Another way to enable the serial-port receive interrupt is to set the bit directly:

```
PIE1bits.RCIE = 1;
```

If not using interrupt priority (IPEN = 0), set these bits:

```
// Disable using interrupt priority.

RCONbits.IPEN = 0;

// Enable all unmasked interrupts.

INTCONbits.GIE = 1;

// Enable all unmasked peripheral interrupts.

INTCONbits.PEIE = 1;
```

If using interrupt priority (IPEN = 1) and the serial interrupt is high priority, set these bits:

```
// Enable using interrupt priority.

RCONbits.IPEN = 1;

// Configure the receive interrupt as high priority.

IPIR1bits.RCIP = 1;

// Enable all high-priority interrupts.

INTCONbits.GIEH = 1;
```

This code sets up an ISR to respond to received serial-port data:

```
// Interrupt vector for high-priority interrupts.

#pragma code high_vector = 0x8

void interrupt_at_high_vector (void)
{
	// Define the function to execute on an interrupt.

	_asm goto high_isr _endasm
}

// The ISR function.

#pragma interrupt high_isr

void high_isr(void)
{
	if ((PIR1bits.RCIF == 1) && (PIE1bits.RCIE == 1))
	{
		// The serial port has received a byte.
		// Place code to read and respond to received data here.

		// Reset the ISR flag.

		PIR1bits.RCIF = 0;
	}
	// Check the interrupt flags for any other enabled interrupts.
}
```

Using Flow Control

Microcontrollers typically require firmware support for hardware flow control. When the firmware has data to send, a hardware interrupt or firmware polling can detect when the flow-control input is in the required state. When receiving data, firmware can set the flow-control output as needed to prevent overruns.

To use hardware flow control on a hardware serial port, select any two otherwise unused port bits and configure one bit as an input and the other as an output.

Sending Data

When sending data, firmware can read the flow-control input and wait if needed before writing a byte to TXREG. In normal operation, the waits should be short, but it's always possible the remote computer will malfunction and leave its flow-control output de-asserted for a long time.

For efficient operation, firmware should minimize the time devoted to waiting for the flow-control input to change state and provide a way to bail out if the wait goes on too long. Tools that can help include timers, task loops, and hardware interrupts.

Firmware with data to send can start a timeout timer and end the wait if the flow-control input isn't asserted after the timer times out. This approach can work well if the waits are normally short or infrequent.

In a task loop, the device performs communications and other tasks as needed in an endless loop. On each pass through the loop, a device with data to send reads the flow-control input. If the input isn't asserted, firmware moves on to the next task. If the input is asserted, the firmware writes the data to the port and performs any other related actions before moving on to the next task. Task loops can work well for many applications.

A hardware interrupt can detect a rising or falling edge on the flow-control input. Firmware can use the ISR to send data when the flow-control pin has indicated that the remote interface is ready to receive data. An interrupt can give the quickest response to changes at a flow-control input.

The PIC18F4520 supports three hardware interrupts on Port B: bit 0 (INT0), bit 1 (INT1), and bit 2 (INT2). Three registers contain bits that monitor or control the interrupts.

Interrupt Control Register (INTCON)

Bit 1 (INT0IF) is set to 1 when INT0 interrupts. Firmware must clear the bit.

Bit 4 (INT0IE) enables INT0.

Bits 6 (GIE) and bit 7 (PEIE) enable the interrupts as described earlier in this chapter.

Interrupt Control Register 2 (INTCON2)

Bit 0 (INTEDG0), bit 1 (INTEDG1), and bit 2 (INTEDG2) configure their corresponding interrupts to generate interrupts on the rising (1) or falling (0) edges on the pins.

Interrupt Control Register 3 (INTCON3)

Bit 0 (INT1IF) and bit 1 (INT2IF) are set to 1 when their corresponding external interrupts occur. Firmware must clear the bits.

Bit 3 (INT1IE) and bit 4 (INT2IE) enable their corresponding external interrupts.

Bit 6 (INT1IP) and bit 7 (INT2IP) set the interrupt priority for their corresponding interrupts. INT0 is always high priority.

Receiving Data

As explained earlier in this chapter, the PIC18F4520 has a 2-byte receive buffer (RCREG) and a shift register (RSR) that stores the bits of an incoming byte. To prevent overrun errors, firmware should de-assert the flow-control output when both of the following are true: RCREG contains a byte, and the Stop bit of the next byte might arrive at the RX pin before firmware can read RCREG.

For example, assume that the bit rate is 9600 bps with 8 data bits and 1 Stop bit, and RCREG contains 1 byte. Another byte can arrive at the RSR and be stored in RCREG in about 1 millisecond. If the flow-control output remains asserted at this point, the remote computer will assume it's OK to send a another byte and may do so. On receiving the Stop bit of the next byte, if both previously received bytes remain in RCREG, the RSR is unable to copy the new byte into RCREG, the microcontroller generates an overrun error, and the recently arrived byte is lost. Note that a slower bit rate gives the receiving computer more time to retrieve received bytes from RCREG.

Many applications never need to de-assert the flow-control output. If firmware can read RCREG at intervals shorter than the time required to transmit three words, the buffer will never overflow.

Application Example

The example that follows demonstrates how to use hardware flow control. On receiving a byte, the firmware de-asserts its flow-control output to tell the remote computer not to send more data while the firmware is preparing its response to the received byte. If the received byte is a code for a lower-case text character (a–z), the receiving computer sends back the character converted to upper case. Otherwise, the receiving computer sends back the same value received. In either case, firmware waits for its flow-control input to be asserted before sending the response.

PBP

```
data_ready_to_send          var     bit
flow_control_input          var     PORTB.4
flow_control_output         var     PORTB.5
serial_in                   var     byte
serial_out                  var     byte

' Configure the flow-control pins as one input and one output.

TRISB.4 = 1
TRISB.5 = 0

' Assert the flow-control output to enable receiving data on the serial port.

flow_control_output = 0
```

This task loop calls two routines in an endless loop:

```
loop:
    gosub receive_serial_data
    gosub transmit_serial_data
GOTO loop
```

This routine checks for a new received byte on the serial port.

```
receive_serial_data:

    if (PIR1.5 = 1) then

        ' A byte has arrived at the serial port. Read it.

        hserin [serial_in]

        ' Tell the remote computer to stop sending data .

        flow_control_output = 1

        ' Set serial_out to a character code to send on the serial port.

        if ((serial_in > 96) and (serial_in < 123)) then

        ' The received character code was lower case (a-z).
        ' Send back the character in upper case.

            serial_out = serial_in - 32
```

```
                else
                        ' For other characters, send back the same character.

                        serial_out = serial_in
                endif

                ' Set a variable to indicate that a byte is ready for transmitting.

                data_ready_to_send = 1
        endif
return
```

This routine checks the data_ready_to_send variable, which indicates whether serial_out contains a byte to be sent on the serial port. If data_ready_to_send = 1 and flow_control_input = 0, the routine sends the byte and sets flow_control_output = 1 to enable receiving another byte.

```
transmit_serial_data:

        if (data_ready_to_send = 1) then

                ' A byte is ready to send on the serial port.

                if (flow_control_input = 0) then

                        ' The flow control input is asserted.
                        ' Reset the variable that indicates a byte is ready to send
                        ' and send the byte.

                        data_ready_to_send = 0
                        hserout [serial_out]

                        ' Tell the remote computer it's OK to send more data .

                        flow_control_output = 0
                endif
        endif
return
```

C18
```
#define flow_control_input   PORTBbits.RB4
#define flow_control_output PORTBbits.RB5

// Configure the flow-control pins as one input and one output.

TRISBbits.TRISB4 = 1;
TRISBbits.TRISB5 = 0;

byte data_ready_to_send = 0;

// Assert the flow-control output to enable receiving data on the serial port.

flow_control_output = 0
```

This task loop calls two routines in an endless loop:

```
 while(1)
{
        receive_serial_data();
        transmit_serial_data();
}
```

This function looks for a new received byte on the serial port. If a byte has arrived, the routine reads it.

```
void receive_serial_data(void)
{
        if (PIR1bits.RCIF == 1)
        {
                // A byte has been received. Read it.

                serial_in = getcUSART();

                // Tell the remote computer to stop sending data .

                flow_control_output = 1;

                // Set serial_out to a character code to send on the serial port.

                if ((serial_in > 96) & (serial_in < 123))
                {
                        // The received character code was lower case (a-z).
                        // Send back the character in upper case.

                        serial_out = serial_in - 32;
                }
```

```
                else
                {
                        // For other characters, send back the same character.

                        serial_out = serial_in;
                }
                // Set a variable to indicate that a byte is ready for transmitting.

                data_ready_to_send = 1;
        }
}
```

This function checks the data_ready_to_send variable, which indicates whether serial_out contains a byte to be sent on the serial port. If data_ready_to_send = 1 and flow_control_input = 0, the function sends the byte and sets flow_control_output = 1 to enable receiving another byte.

```
void transmit_serial_data()
{
        if (data_ready_to_send == 1)
        {
                // A byte is ready to send.
                // If the flow control input is asserted, send the byte.
                // Otherwise wait until the routine is called again.

                if (flow_control_input == 0)
                {
                        data_ready_to_send = 0;
                        while(BusyUSART());
                        putcUSART(serial_out);

                        // Tell the remote computer it's OK to send more data .

                        flow_control_output = 0;
                }
        }
}
```

Adding Ports

The options for adding serial ports to an embedded system include using a microcontroller that has the needed number of ports, using firmware UARTs, and interfacing to external UARTs.

Multiple On-chip UARTs

Hardware UARTs handle much of the burden of serial communications. If your microcontroller has an available hardware UART, it makes sense to use it. For applications that need multiple serial ports, some microcontrollers, including the PIC18F8722 and other PIC microcontrollers, have multiple hardware UARTs. The Rabbit 4000 microprocessor from Rabbit Semiconductor Inc. (*www.rabbit.com*) has six on-chip asynchronous serial ports.

Firmware UARTs

Another way to implement serial communications is via a firmware UART such as the ports supported by the PICBASIC PRO and C18 compilers. These ports can use any generic I/O pins for serial communications. Firmware ports can have features not available on a hardware port. For example, PICBASIC PRO's firmware ports can use serin2 and serout2 statements with an optional FlowPin parameter that specifies a pin to use for flow control. When using flow control, a serin2 statement sets its FlowPin output to enable receiving data. A serout2 statement waits for its FlowPin input to be asserted before sending data. An optional timeout value causes the code to jump to a label if the pin isn't asserted within the specified time. In most cases, however, a hardware UART's capabilities are more useful than the added features of a firmware UART.

External UARTs

Another way to add a serial port is via an interface to an external UART.

Components

The 16550 UART used in PCs for many years has eight data pins plus various address, status, and control pins. Many microcontrollers don't have enough port pins to interface to this and similar chips. An alternative is a UART with a synchronous serial interface.

An SPI/Microwire UART

A chip that requires as few as three lines for a bidirectional link is Maxim's MAX3100 SPI/Microwire UART. The chip converts between synchronous and asynchronous serial data (Figure 11-2). The synchronous data is compatible with SPI and Microwire, which require a clock line plus a data line for each direction.

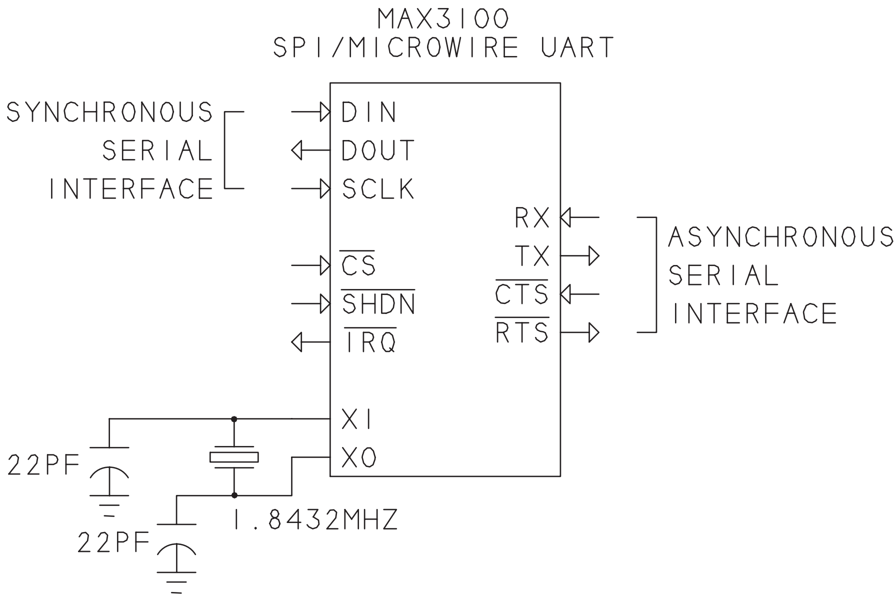


Figure 11-2: The MAX3100 converts between SPI or Microwire serial data and asynchronous format.

A microcontroller can communicate with the MAX3100 by sending and receiving synchronous data. The microcontroller can toggle SCLK as needed without worrying about maintaining a specific bit rate. The only restrictions on SCLK's frequency are a minimum pulse width (high or low) of 100 ns and a minimum clock period (high + low) of 238 ns.

The synchronous interface exchanges 16-bit words. Eight bits are data, and the other bits can hold status and control information. The chip can also send and receive configuration data.

A crystal or ceramic resonator provides the timing reference for the UART's bit-rate generator. A 1.8432MHz crystal supports bit rates from 300 to 115,200.

On receiving synchronous data to transmit, the MAX3100 writes a Start bit, the data bits, a parity bit if used, and a Stop bit to TX. The data can have 7 or 8 bits plus an optional parity bit. The MAX3100 doesn't calculate a value for a parity bit so the sending device must set the parity as desired. The Max3100 can generate an interrupt request on receiving a parity bit of 1.

The chip has an 8-byte FIFO for each direction, so the microcontroller or other CPU doesn't have to read each received byte from the UART before the next byte arrives.

This page intentionally left blank

12

Network Programming

When three or more devices share a communication path, serial-port programming becomes more complicated. Each computer, or node, in the network, needs to know when it's OK to transmit and whether to respond to or ignore a received message. This chapter shows how to ensure reliable and efficient communications in a serial network and how to implement a basic network protocol.

Managing Traffic

One of the first things to decide is how the network will manage its traffic. A typical network has the following features:

- Each node can send and receive data.
- Only one node transmits at a time.
- Each node recognizes and responds to messages intended for it and ignores all others.
- The transmitting node detects when a node didn't receive a message and takes appropriate action.

Steps in Exchanging a Message

In even the most basic networks, all messages must arrive at their intended destinations without errors, and each node must respond only to those messages intended for it.

For example, assume that Node A wants to send a message to Node B, telling Node B to set a port to a value and to send back a value read from another port. In a typical serial network, all of the following must take place:

Node A does the following:

Enables its network driver.
Sends the address of the node to transmit to.
Sends the message.
Disables the network driver and waits for a response.

Node B does the following:

Reads incoming data.
Detects the node's address.
Reads the message associated with the address.
Detects when the message has ended.
Takes the requested action.
Prepares a response.
Enables the network driver.
Sends the response.
Disables the network driver.

Node A then does the following:

Reads the response.
Takes any required action.

At the same time, all of the other nodes must do the following:

Read incoming data.
Detect the transmitted address and ignore the message associated with the address.

Protocols

The nodes in a network must agree on a protocol for managing communications. Three types of network protocol are primary/secondary, token passing, and collision detecting.

Primary/Secondary

A primary/secondary protocol, also called the master/slave protocol, is often the least complex network protocol a network can implement. One computer is designated the primary node in charge of controlling all network traffic. The other computers are secondary nodes that respond to communications from the primary node. A network might have a PC as the primary node and embedded systems as secondary nodes.

To give each node a chance to communicate, the primary node can poll, or send a message to, each of the secondary nodes in sequence. Each poll can request a response, which might contain an acknowledgment, requested data, an error message, or other information. A secondary node transmits only when the primary node has requested a response. Any message from one secondary node to another must pass through the primary node.

The main limitation of the protocol is the delays that occur as each node waits to be polled. For a critical alarm system, waiting to be polled could be a problem, while a data-acquisition system might tolerate long delays.

Token Passing

A token-passing protocol allows any node to obtain control of the network. The node in control is said to have the token.

The network protocol defines how a node knows if it has the token and how to pass the token to another node. The token can be a defined bit or variable that each node sets to indicate whether or not the node holds the token.

A node that wants to pass the token to another node gives up the token it has (by clearing its token bit, for example) and sends a message to inform another node that it now has the token. This node then takes whatever action it wants and passes the token on. The nodes can pass the token in a defined sequence or use another method to decide who gets the token next.

This protocol enables any node that has the token to talk directly to another node. But a node that doesn't have the token can't interrupt with an emergency message.

Collision Detecting

A collision-detecting protocol allows any node to transmit whenever the transmission path is free. If two or more nodes try to transmit at the same time, all of the nodes (or with some protocols, all nodes but one), must detect the collision,

stop transmitting, and try again after a delay. This protocol is useful when any node needs to be able to transmit whenever it wants and when the overall network traffic is light enough that delays due to collisions are infrequent. Ethernet and I^2C use collision-detecting protocols.

To use the protocol, the nodes must be capable of detecting collisions, and the drivers must be able to survive multiple drivers enabled at the same time, if only briefly.

One way to detect a collision is for each transmitting node to attempt to read back what it sends. If the data read matches the data written, the node assumes there was no collision and the transmission succeeded. If the data read doesn't match the data written, the node assumes that another node is trying to transmit and tries again later. Different nodes should use different delay times, either by assigning each a different, fixed delay or by using random values. Otherwise, the nodes will all retry at the same time and no one will ever get through. The receiving nodes must also recognize and ignore failed attempts at transmitting.

Collision-detecting protocols typically aren't practical for RS-485 networks. With the UARTs used for most RS-485 interfaces, the nodes aren't able to examine each bit as it arrives, so bit-by-bit collision detecting isn't possible.

If two or more drivers are enabled, RS-485 chips have protection circuits that limit the current and eventually disable the outputs, but the currents can be as high as 250 mA before the output is disabled. These safety features are useful for protecting the circuits during occasional malfunctions, but a protocol that routinely causes these high currents wastes power and can stress the circuits.

In contrast, open-collector and open-drain logic used in I^2C synchronous communications can have multiple drivers enabled without drawing large currents, and the software is often capable of bit-by-bit monitoring.

Using Existing Protocols

One way to get network programming up and running is to use an existing protocol. Various organizations and companies have developed protocols used in serial communications.

A fieldbus is a digital communications link developed for monitoring and control systems. These are some examples of standard fieldbuses used in RS-485 networks:

- BACnet (Building Automation Control network). For building automation and other monitoring and control applications. From the American Society

of Heating, Refrigerating, and Air-Conditioning Engineers (ASHRAE) (*www.ashrae.com*).

- BITBUS. Introduced by Intel. General-purpose primary/secondary protocol. BITBUS European Users Group (*www.bitbus.org*) has documentation. An expanded version is defined by the withdrawn IEEE Standard 1118-1990.
- CTNet. Features de-centralized control and high performance (*www.controltechniques.com*).
- INTERBUS. Used in automotive applications. Promoted by the INTERBUS Club (*www.interbusclub.com*).
- Modbus. Developed by Modicon and now an open standard. Modbus RTU transmits binary data. Modbus ASCII transmits text data. Promoted by Modbus-IDA (*www.modbus.org*)
- PROFIBUS (Process Fieldbus). Used in manufacturing, process control, and building automation. From Profibus International (*www.profibus.org*).
- SAE J1708. For vehicle applications. From the Society of Automotive Engineers (*www.sae.org*).

High Tech Horizon *(www.hth.com*) offers the free, generic S.N.A.P. protocol. Some data-acquisition devices support vendor-specific ASCII command sets.

Debugging Tips

Because of their complexity, networks can be challenging to debug. A gradual and deliberate approach to putting together or debugging a network can save time in the long run. These are some techniques that can be useful:

Start with two nodes. In a primary/secondary network, connect the primary node to one secondary node and get that working before adding more nodes.

Keep the distance short. If possible, place the nodes in the same room, side-by-side, until things are working.

Use a slow bit rate. A slow bit rate is especially useful if your debugging tools are limited. At 300 bps, you can watch LEDs flicker on a breakout box when a node transmits.

Use your programming environment's debugging capabilities. Breakpoints, watch variables, single-stepping, and other debugging tools can help isolate the source of problems.

Monitor the network traffic. As with RS-232, a digital oscilloscope or logic analyzer can help by showing exactly what's happening on the line. Many

scopes have a math function that enables viewing a differential voltage as the difference between the voltages on the two channels.

Humble components such as LEDs and switches can also be useful. On a microcontroller, interface LEDs to spare output bits and write code that toggles the bits to indicate events. For example, firmware can turn on an LED when the node recognizes its address and turn on another LED after receiving a message. Add toggle or slide switches to spare input port bits and have the program send the values of the bits in its messages.

Addressing

In a typical network, each node has an assigned address and each message contains the address of the recipient. On receiving a message, a node must detect the address to determine whether to process or ignore the message.

Assigning Addresses

A node address is a value unique to the node and can be any number of bits. Some networks use addresses that correspond to ASCII codes.

If there are fewer than 128 nodes, you don't need 8 bits to specify the node and can get the most use out of a transmitted byte by assigning extra bits to other uses. For example, in a 16-node network, bits 0–3 can specify the node number, with bits 4–7 holding a command or other information.

Detecting Addresses

One challenge in sending addresses is that the nodes have to distinguish between addresses and other information. For example, imagine a network where each transmitted message begins with a byte containing the address of the recipient. On recognizing its address, a node knows that the bytes that follow are intended for it.

If Node 05h receives the byte 03h followed by 05h, how does Node 05h know whether the second byte is part of a message meant for another node or an address that begins a new message?

There are several ways to distinguish between addresses and other data:

- The addresses can use a reserved set of values that messages never use.
- The network can define a message format that specifies where the address is stored in messages.

- Communications can dedicate one data bit to indicate whether the other data bits contain an address or data.

Reserving Address Values

Reserving a set of values for use only as addresses makes it easy to distinguish addresses from data with the limitation that messages can't uses the reserved values for other purposes.

If the messages contain text characters (which can include characters for numerals), network addresses can use any values that don't represent characters. For example, if the messages use only US ASCII (codes 0–127), addresses can use the values 128–255.

If the messages contain binary data, one solution is to send the data in ASCII Hex format as described in Chapter 2. Because ASCII Hex can represent any binary value using just 16 codes, plenty of codes remain for use as addresses.

Defining a Message Format

Many network protocols define a message format with the address and other information in assigned locations in the message.

For example, an 8-byte message might consist of an address byte followed by seven message bytes. On receiving a byte, a node examines the value to see if it matches the node's address. If it's a match, the node reads and acts on the seven bytes that follow, then waits for another address byte to examine. If the address doesn't match, the node counts but otherwise ignores the seven bytes that follow.

With this method, every node has to detect every byte sent, if only to know when the message is finished. A node that misses a byte for any reason won't detect the correct address bytes in future messages.

Using Dedicated Codes for Start and End

To make it easier to detect addresses, a protocol can define values that indicate Start of Transmission and End of Transmission. A node that gets lost can then recover on the next Start of Transmission code. Conventional values are 02h (Control+B) for Start of Transmission and 03h (Control+C) for End of Transmission. Some networks use other characters such as ":" or "$". These values are then unavailable for other uses, so using dedicated codes is useful mainly for communications that send data as plain text.

A message format can also define a field that indicates the length of the data that follows, and receiving nodes can use this value to determine when a message ends.

Headers

A header is information that uses a defined format and appears at the beginning of a message or other block of data. The header typically consists of a series of fields, with each field having a defined size and location. The information included in a header can vary. A header might include some or all of these items:

- Start-of-communication code.
- Address of the receiving node.
- Address of the sending node.
- Length of the data that follows the header.
- Checksum or other value used for error detection.
- Description of the type of data that follows.
- Time and date information.

9-bit Format

Another option for distinguishing between addresses and data uses a 9-bit format. Bit 8 (the ninth bit) indicates whether bits 0–7 contain data (0) or an address (1). Not all UARTs support the 9-bit format. The longer words in 9-bit data mean that communications are slightly more sensitive to mismatches in the rates at the transmitting and receiving ports. Messages that consist of only US ASCII text can use seven data bits for a text character and the eighth bit to indicate address or data.

Embedded Systems

Some microcontrollers have a built-in ability to use the ninth bit to detect an address. The PIC18F4520 introduced in Chapter 11 is an example.

To transmit 9-bit data, PIC18F4520 firmware must set the TX9 bit in the TXSTA register to 1. Before writing a byte to transmit to TXREG, firmware should write the ninth bit to the TX9D bit in TXSTA.

To receive 9-bit data, firmware must set the RCSTA register's RX9 bit to 1. Before reading RCREG to obtain a received byte, firmware should read the RCSTA register's RX9D bit to obtain the ninth bit.

Firmware is responsible for determining the value of the ninth bit to transmit and interpreting the value of a received ninth bit. The hardware has no built-in support for setting and detecting address/data bytes. Firmware must also define an 8-bit address for the device.

To use address detecting with 9-bit data, in the RCSTA register, set ADDEN = 1. A computer that wants to communicate with the device must begin by sending a 9-bit word consisting of the device's address with the ninth (most significant) bit = 1. In the PIR1 register, RCIF will be set only after receiving a word whose ninth bit = 1. When RCIF = 1, firmware should read the received byte. If the byte matches the device's address, firmware sets ADDEN = 0 to enable reading data. All received words that follow with bit 9 = 0 contain data intended for the device. On receiving a word with bit 9 = 1, firmware should again check for a matching address. If it's not a match, firmware should set ADDEN = 1 to disable setting RCIF on any data that follows.

Other microcontrollers, such as Maxim Integrated Products, Inc.'s DS89C420, can store a node address and a broadcast address and respond only when a received address matches one of these values.

PCs

The UARTs in PCs typically don't have full hardware support for 9-bit protocols, but applications can use Mark and Space parity to implement a software-assisted 9-bit protocol.

Before changing the parity type, an application should ensure that all data written with the previous parity setting has transmitted. Chapter 6 described methods for finding out when the data has finished transmitting to determine when it's OK to disable an RS-485 driver. The same methods are also useful for determining when it's OK to change the parity type. An application that sends 9-bit data might also need to add delays to ensure that the receiving computer processes received bytes in the correct order, as described below.

To use 9-bit data in a .NET application, set the SerialPort object's DataBits property to 8 and set the ParityReplace property to zero to disable the parity-replace feature. The ParityReplace property can specify a byte that will replace any byte received with a parity error. With 9-bit data, you want to preserve the data rather than overwrite it.

VB

```
myComPort.DataBits = 8
myComPort.ParityReplace = 0
myComPort.Open
```

VC#
```
myComPort.DataBits = 8;
myComPort.ParityReplace = 0;
myComPort.Open();
```

To write to a node, select Mark parity, write the node's address, and then select Space parity and write the data intended for that node.

VB
```
' Write a node's address.

myComPort.Parity = Parity.Mark
myComPort.Write("h")

' Delay as needed to ensure that the address has transmitted.
' Write data to the node.

myComPort.Parity = Parity.Space
myComPort.WriteLine("Test data for node h.")

' Delay as needed to ensure that the data has transmitted.
' Write a node's address.

myComPort.Parity = Parity.Mark
myComPort.Write("m")

' Delay as needed to ensure that the address has transmitted.
' Write data to the node.

myComPort.Parity = Parity.Space
myComPort.WriteLine("Test data for node m.")
```

VC#

```
// Write a node's address.

myComPort.Parity = Parity.Mark;
myComPort.Write("h");

// Delay as needed to ensure that the address has transmitted.
// Write data to the node.

myComPort.Parity = Parity.Space;
myComPort.WriteLine("Test data for node h.");

// Delay as needed to ensure that the data has transmitted.
// Write a node's address.

myComPort.Parity = Parity.Mark;
myComPort.Write("m");

// Delay as needed to ensure that the address has transmitted.
// Write data to the node.

myComPort.Parity = Parity.Space;
myComPort.WriteLine("Test data for node m.");
```

When a byte with a parity error is the next byte to be read from the port, the ErrorReceived event fires with an EventType of SerialError.RXParity. Chapter 10 showed how to assign a method to the ErrorReceived event.

A computer receiving data waits for a parity error and then reads the byte with the error. If the byte matches the node's address, the application accepts and uses all data that follows until the next parity error. If a byte with a parity error doesn't match the node's address, the application reads all data that follows up to the next parity error but doesn't use the data in any other way.

An application can set a variable (acceptData in the example below) that indicates whether the most recent address was a match (True) or not (False):

VB

```
Dim acceptData as Boolean
Dim myAddress As Integer = Microsoft.VisualBasic.AscW("h")

Private Sub ErrorReceived _
    (ByVal sender As Object, ByVal e As SerialErrorReceivedEventArgs)

    Dim SerialErrorReceived1 As SerialError

    SerialErrorReceived1 = e.EventType

    Select Case SerialErrorReceived1

    Case SerialError.RXParity

        If (myComPort.ReadChar = myAddress) Then
            acceptData = True
        Else
            acceptData = False
        End If

    End Select
End Sub
```

VC#
```
internal bool acceptData;
internal int myAddress = (int)'h';

private void ErrorReceived( object sender, SerialErrorReceivedEventArgs e )
{

        SerialError SerialErrorReceived1 = 0;
        SerialErrorReceived1 = e.EventType;

        switch ( SerialErrorReceived1 )
        {
            case SerialError.RXParity:

                if ( ( SelectedPort.ReadChar() == myAddress ) )
                {
                    acceptData = true;
                }
                else
                {
                    acceptData = false;
                }
                break;
        }
}
```

A DataReceived event can check the state of acceptData before deciding what to do with new received data. Chapter 10 showed how to assign a method to the DataReceived event.

VB
```
Friend Sub DataReceived _
    (ByVal sender As Object, ByVal e As SerialDataReceivedEventArgs)

    Dim newReceivedData As String

    newReceivedData = selectedPort.ReadExisting

    If acceptData Then

        ' The received address matches our address.

    End If

End Sub
```

VC#
```
internal void DataReceived( object sender, SerialDataReceivedEventArgs e )
{
    string newReceivedData = null;

    newReceivedData = SelectedPort.ReadExisting();

    if ( acceptData )
    {
        // The received address matches our address.
    }
}
```

Using events to detect received data and addresses assumes that the events are raised in the order they occur. However, .NET doesn't guarantee that the DataReceived and ErrorReceived events will fire in order. For example, if the transmitting computer sends an address immediately followed by data, the DataReceived event might fire before the ErrorReceived event and cause the receiving computer to think the data is for a different address. To prevent problems, the transmitting computer can insert a delay either before or after changing parity. The delay allows time for the receiving computer to process an address before the data arrives and time to process received data before another address arrives.

To simplify the programming, a primary/secondary network can use the address bit only when the primary node transmits to a secondary node. If the secondary nodes transmit only in response to received communications, the primary node can assume that any incoming data is from the node most recently addressed. The other secondary nodes ignore data sent by the other secondary nodes because a matching address wasn't detected.

13

An RS-485 Network

This chapter presents a basic network that uses a primary/secondary protocol. A primary node sends commands to the secondary nodes. Each secondary node detects, reads, and responds to commands directed to it.

You can use the network as a base for designing projects using PCs and embedded systems of any type, in any combination. The example code includes PC code for the primary node and microcontroller code for the secondary nodes.

Connecting the Nodes

Chapter 12 introduced the options for wiring the nodes in an RS-485 network. Figure 13-1 shows a configuration you can use for the example network in this chapter.

Transceivers

The network can use any appropriate transceivers for half-duplex RS-485 ports, including high-speed, low-speed, low-voltage, short-distance, and isolated components. A PC can use an expansion card with an RS-485 interface, a

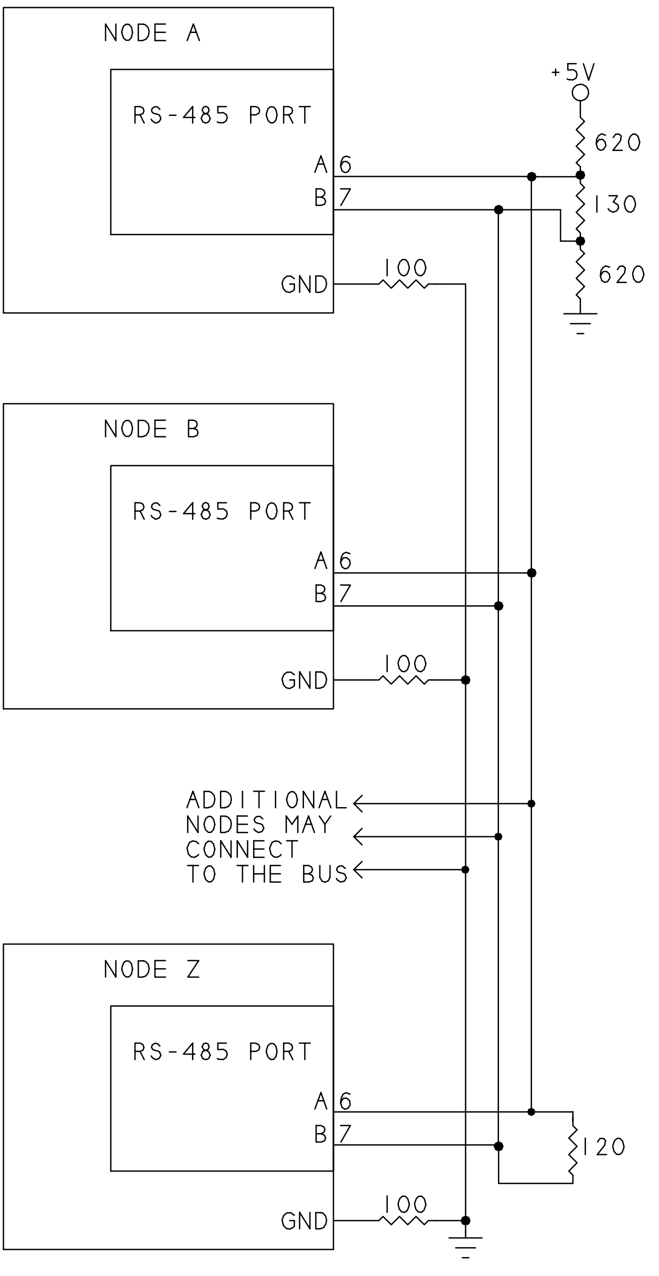


Figure 13-1: The RS-485 network can use this wiring to connect the nodes.

USB/RS-485 converter, or an RS-232/RS-485 converter that connects to an RS-232 port. Microcontroller ports can interface to RS-485 transceivers.

To control the driver-enable inputs, the nodes can use automatic driver-enable circuits as described in Chapter 7 or software-controlled driver-enable lines. The methods used to control the driver-enable lines determine in part whether the nodes need to insert delays before sending or responding to commands.

Terminating and Biasing

At one end of the network, three resistors in series hold the inputs high when no drivers are enabled on the network. The 130Ω resistor provides a parallel line termination and the two 620Ω resistors provide biasing. At the other end of the network, a 120Ω resistor provides a parallel termination. If the line is electrically short as defined in Chapter 7, you don't need terminating or biasing resistors.

Cabling

For the network wires, unshielded twisted pair works well. Include a ground wire to each node unless you're sure that all nodes already have a common signal-ground connection.

Example Protocol

The example network uses a command/response protocol. You can expand on the protocol as needed, adding additional commands, error codes, error checking of received communications, or other features.

Addresses

Each secondary node has a unique assigned address from 97 to 122, which correspond to ASCII codes for the characters a–z. The primary node doesn't have an address. Each command has a value from 48 to 57, which correspond to ASCII codes for the characters 0–9.

Message Format

Messages from the primary node use this format:

Byte 0 is a ":" character that signals the start of a message.

Byte 1 is the address of the node being addressed.

Byte 2 is a command code.

Additional bytes can contain data specific to the command. Some commands have no command-specific bytes.

The final byte is a line-feed code (0Ah) that signifies the end of the message. A CR code (0Dh) can precede or follow the LF but isn't required.

On receiving a message from the primary node, a secondary node returns a message with this format:

Byte 0 is a ":" character that signals the start of a response.

Byte 1 is the address of the node returning the response.

Additional bytes can contain data specific to the command. Some responses have no command-specific bytes.

The final byte is a line-feed code that signifies the end of the message. A CR code can precede the LF but isn't required.

Commands

The code example in this chapter implements two commands: read_byte and write_byte. On receiving a valid command, a node returns a response. A node that receives an unsupported command or detects an error returns no response.

Reading a Byte

The read_byte command requests a byte of data from a specified location in a secondary node. The location is an application-defined byte that can indicate a port or variable:

Byte Number	Contents (ASCII/ASCII Hex)	Description
0	: (ASCII 58)	Start-of-communication indicator
1	address	Identifies the recipient
2	1 (ASCII 48)	read_byte command code
3	location	Identifies the location to read
4	LF (ASCII 10)	End-of-communication indicator

The response to read_byte contains the requested byte in ASCII Hex format:

Byte Number	Contents (ASCII/ASCII Hex)	Description
0	: (ASCII 58)	Start-of-communication indicator
1	address	Identifies the sender
2	data	Requested data, most significant nibble (ASCII Hex format)
3	data	Requested data, least significant nibble (ASCII Hex format)
4	LF (ASCII 10)	End-of-communication indicator

A command sent to node h to read a byte from location b is:

```
:h1b
```

followed by a LF.

These are the transmitted values in hexadecimal:

```
3a 68 31 62 0a
```

On accepting the command, node h reads location b. If the location contains the value A5h, the node responds with:

```
:ha5
```

followed by a LF.

These are the values transmitted in hexadecimal:

```
3a 68 61 35 0a
```

Writing a Byte

The write_byte command writes a byte of data to a specified location in a secondary node. The location is an application-defined byte that can indicate a port or variable. The byte is sent in ASCII Hex format:

Byte Number	Contents (ASCII/ASCII Hex)	Description
0	: (ASCII 58)	Start-of-communication indicator
1	address	Identifies the recipient
2	2 (ASCII 49)	write_byte command
3	location	Identifies the location to write to
4	data	Data to write, most significant nibble (ASCII Hex format)
5	data	Data to write, least significant nibble (ASCII Hex format)
6	LF (ASCII 10)	End-of-communication indicator

The response to write_byte identifies the sender and serves as an acknowledgement that the command was received:

Byte Number	Contents (ASCII/ASCII Hex)	Description
0	: (ASCII 58)	Start-of-communication indicator
1	address	Identifies the sender
2	LF (ASCII 10)	End-of-communication indicator

A command sent to node i to write the byte FEh to location c is:

```
:i2cfe
```

followed by a LF.

These are the transmitted values in hexadecimal:

```
3a 69 32 63 66 65 0a
```

On accepting the command, node i writes the value FEh to location c and responds with:

```
:i
```

followed by a LF.

These are the values transmitted in hexadecimal:

```
3a 69 0a
```

Polling the Nodes

On a PC that functions as the primary node, an application can send commands, log received data, and take other actions as needed. Application code can control the RS-485 transceiver's driver-enable line if needed. If the RS-485 interface uses automatic driver-enable control, you can use terminal-emulator software to access the nodes by typing commands and viewing responses.

The example code below supports both automatic and software-controlled driver-enable lines. With automatic control, the transceiver's receiver-enable input connects to the driver-enable input to disable the receiver while the PC is transmitting. With software control, the RS-485 transceiver's receiver-enable input is tied low so the PC can read back the data it sends and disable the driver on detecting that all of the data has transmitted. (See Chapter 6 for more about driver-enable circuits.)

The routines below assume myComPort is an open SerialPort object.

Configuring the Driver-enable Line

For circuits with software driver-enable control, the routine below sets the state of the driver-enable output. As shown, the routine uses the RtsEnable property to control the RTS. If the driver-enable line is DTR, change the code to use the DtrEnable property instead.

VB

```
Private Sub Rs485DriverEnableControl(ByVal driverDisable As Boolean)

    If driverDisable Then
        myComPort.SelectedPort.RtsEnable = False
    Else
        myComPort.SelectedPort.RtsEnable = True
    End If
End Sub
```

VC#

```
private void Rs485DriverEnableControl(bool driverDisable)
{
	if (driverDisable)
	{
		myComPort.RtsEnable = false;
	}
	else
	{
		myComPort.RtsEnable = true;
	}
}
```

Sending Commands

The Send_Command routine accepts command parameters, sends a command, and displays a received response.

VB

```
Private driverDisable As Boolean

' Uncomment one of these lines to indicate the method of controlling the RS-485
' driver-enable line.

Private softwareDriverEnable As Boolean = False ' software control
'Private softwareDriverEnable As Boolean = True ' hardware control

Private Sub Send_Command(ByVal node As String, ByVal command As String, _
	ByVal address As String, ByVal data As String)

	Dim dataSent As String
	Dim response As String

	' Define LF as the NewLine character for serial data.

	myComPort.NewLine = System.Convert.ToString((char)10)

	If softwareDriverEnable Then
		Rs485DriverEnableControl(True)
	End If

	' Write a command to the serial port.

	myComPort.WriteLine(":" + node + command + address + data)
```

```
        If softwareDriverEnable Then

            ' Read back what was transmitted to ensure that all of the data
            ' has gone out.
            ' Then disable the RS-485 driver.
            ' The RS-485 transceiver's receiver must be permanently enabled
            ' in hardware.

            dataSent = myComPort.ReadLine()
            Rs485DriverEnableControl(False)

        End If

        ' Read and decode the response.

        response = myComPort.ReadLine()

        If response.StartsWith(":") Then
            If response.Substring(1, 1) = node Then

                Select Case command

                    Case "1"
                        ' write_byte command

                        Console.WriteLine _
                        ("write_byte command acknowledged.")

                    Case "2"
                        ' read_byte command. Display the received data.

                        Console.WriteLine("read_byte response data: " + _
                            response.Substring(2, 2))
                End Select
            End If
        End If
    End Sub
```

VC#
```
// Uncomment one of these lines to indicate the method of controlling the RS-485
// driver-enable line.

private bool softwareDriverEnable = true; // software control
//private bool softwareDriverEnable = false; // hardware control

private bool driverDisable;

private void Send_Command( string node, string command, string address, string data )
{
    string dataSent = null;
    string response = null;

    // Define LF as the NewLine character for serial data.

    myComPort.NewLine = System.Convert.ToString((char)10);

    if (softwareDriverEnable)
    {
        Rs485DriverEnableControl(true);
    }
    // Write a command to the serial port.

    myComPort.WriteLine( ":" + node + command + address + data );

    if (softwareDriverEnable)
    {
        // Read back what was transmitted to ensure that all of the data goes out.
        // Then disable the RS-485 driver.
        // The RS-485 transceiver's receiver must be permanently enabled
        // in hardware.

        dataSent = myComPort.ReadLine();
        Rs485DriverEnableControl(false);
    }
    // Read and decode the response.

    response = myComPort.ReadLine( );
```

```
        if (response.StartsWith(":"))
        {
            if (response.Substring(1, 1) == node)
            {
                switch (command)
                {
                    case "1":
                        Console.WriteLine(
                            "write_byte command acknowledged.");
                        break;
                    case "2":
                        Console.WriteLine("read_byte response data: " +
                            response.Substring(2, 2));
                        break;
                }
            }
        }
    }
```

Responding to Polls

A secondary node reads all incoming bytes looking for the ":" character that indicates the start of a message. On receiving ":", the node can check the address right away or wait to receive a LF code. In the first approach, the node reads the next received byte to find out if it matches the node's address. If the byte matches, the node stores the data that follows the address up to a LF and then acts on the received command. If the byte doesn't match the address, the node ignores any received data up to the next ":". In the second approach, the node stores data that follows any ":" up to a LF. After receiving a LF, the node examines the received address and acts on the received command only if the address matches. The code that follows uses the second approach.

Auxiliary Routines

The example firmware for network communications includes a task loop and routines that implement delays and convert between ASCII Hex and binary data.

Definitions

The code supports a firmware-controlled or hardware-controlled driver-enable line. If needed, the firmware can implement a delay before responding to a command.

PBP

```
DEFINE HSER_CLROERR

COMMAND_START               con    ":"
MAX_COMMAND_LENGTH          con    5
MY_ADDRESS                  con    "h"

' Use any spare output port bit to control the driver enable line
' Unneeded if using automatic hardware driver enable:

driver_enable               var    PORTB.3

command_index               var    byte
command_response            var    byte[6]
converted_byte              var    byte
data_ready_to_send          var    bit
delay_before_responding     var    bit
firmware_driver_enable      var    bit
index                       var    byte
lower_nibble                var    byte
network_state               var    byte
received_command            var    byte[6]
response_index              var    byte
serial_in                   var    byte
success                     var    bit
upper_nibble                var    byte
value_to_convert            var    byte

command_index = 0
command_response[0] = COMMAND_START
command_response[1] = MY_ADDRESS
network_state = "r"
response_index = 0

' Output port bit used only for testing:

TRISB.0 = 0
low PORTB.0
```

```
' Uncomment one of these lines.

' delay_before_responding = 0  ' No delay before responding.
delay_before_responding = 1  ' Delay before responding

' Uncomment one of these lines:

' firmware_driver_enable = 0  ' Circuits have automatic driver-enable control.
firmware_driver_enable = 1  ' Firmware controls the driver-enable line.

if (firmware_driver_enable = 1) then

	' Define the driver-enable line and disable the driver.

	TRISB.3 = 0
	driver_enable = 0
endif
```

C18

```
// Comment out this line if using automatic hardware control of the
// RS-485 driver-enable line.

#define firmware_driver_enable

// Comment out this line if the remote computer requires no extra time to disable
// its RS-485 driver before the microcontroller responds to a received command.

#define delay_before_responding

#if defined(firmware_driver_enable)
	#define driver_enable  PORTBbits.RB3
#endif

#define COMMAND_START 58

const byte MAX_COMMAND_LENGTH = 5;

// The microcontroller's network address:

const char MY_ADDRESS = 'h';
```

```
byte            command_index;
char            command_response[6];
char            network_state = 'r';
char            received_command[6];
byte            response_index = 0;
unsigned char serial_in;

// Define the driver-enable line and disable the driver.

#if defined(firmware_driver_enable)
        TRISBbits.TRISB3 = 0;
        driver_enable = 0;
#endif

// Output port bit used only for testing:

TRISBbits.TRISB0 = 0;
PORTBbits.RB0 = 0;
```

Task Loop

A task loop can handle network communications and other activities. In the example below, the serial_communications routine implements a state machine with three states. In state "r", the device is waiting to receive data. In state "d", the device has received a command and is delaying before responding to allow the primary node time to disable its driver. If the primary node disables its driver immediately after transmitting, the firmware can be configured so it never enters state "d". In state "t", the device is ready to transmit data.

PBP
```
loop:

    gosub serial_communications

    ' Add other tasks here.

goto loop

serial_communications:

    select case network_state

        case "r"
            gosub receive_serial_data

        case "d"
            gosub check_response_delay

        case "t"
            gosub transmit_serial_data

        case else

    end select
return
```

C18
```
while(1)
{
	serial_communications();

	// Add other tasks here.
}

void serial_communications(void)
{
	switch (network_state)
	{
		case 'r':
		{
			receive_serial_data();
			break;
		}
		case 'd':
		{
			check_response_delay();
			break;
		}
		case 't':
		{
			transmit_serial_data();
			break;
		}
		default:
			break;
	}
}
```

Implementing Delays

On receiving a valid command, the code calls the prepare_to_respond function, which either starts the delay timer and sets network_state to "d" or sets network_state to "t" as appropriate.

If delay_before_responding is defined, on receiving a valid command, the device starts a timer and serial communications enter the "d" state. On each pass through the task loop, the code calls check_response_delay to find out if the delay time has elapsed. If so, the routine stops the timer and switches to the "t" state.

If delay_before_responding isn't defined, the timer never starts and communications never enter the "d" state.

PBP
```
prepare_to_respond:

        response_index = 0;
        if (delay_before_responding = 1) then
                gosub start_response_delay_timer
                network_state = "d"
        else
                network_state = "t"
        endif
return
```

C18
```
void prepare_to_respond(void)
{
        response_index = 0;
        #if defined(delay_before_responding)
                start_response_delay_timer();
                network_state = 'd';
        #else
                network_state = 't';
        #endif
}
```

The start_response_delay_timer function initializes and starts the timer.

PBP
```
start_response_delay_timer:

        ' This example sets a delay of around 0.5 second assuming FOSC = 4 MHz.

        ' Timer enabled, 16-bit, internal clock, prescaler = 256.

        T0CON = $87

        ' Load the timer with F800h.

        TMR0L = $00
        TMR0H =$F8

        return
```

C18
```
void start_response_delay_timer(void)
{
	// This example sets a delay of around 0.5 second assuming FOSC = 4 MHz.

	OpenTimer0( TIMER_INT_OFF &
			T0_16BIT &
			T0_SOURCE_INT &
			T0_PS_1_256 );
	WriteTimer0(0xf800);
}
```

The check_response_delay function finds out if the delay time has elapsed and if so, changes the network_state variable to enable sending a response:

PBP
```
check_response_delay:
	if (delay_before_responding = 1) then

		if (INTCON.2 = 1) then

			' The delay time has elapsed.
			' Stop the timer and set network_state to enable responding.

			T0CON = 0
			INTCON.2 = 0
			network_state = "t"
		endif
	endif
return
```

C18
```
void check_response_delay(void)
{
	#if defined(delay_before_responding)

		if (INTCONbits.TMR0IF == 1)
		{
			// The delay time has elapsed.
			// Stop the timer and set network_state to enable responding.

			CloseTimer0;
			INTCONbits.TMR0IF = 0;
			network_state = 't';
		}
	#endif
}
```

Converting between Bytes and ASCII Hex

The network encodes data in ASCII Hex format. Chapter 2 showed .NET code for converting between binary and ASCII Hex bytes. Microcontroller code can perform these conversions as well.

The byte_to_ascii_hex function converts a byte (value_to_convert) to two ASCII hex characters that represent the byte's value (upper_nibble and lower_nibble):

PBP

```
byte_to_ascii_hex:

    ' Represent the byte variable value_to_convert as
    ' ASCII Hex characters upper_nibble and lower_nibble.

    upper_nibble = (value_to_convert & $f0) >> 4

    if ((upper_nibble >= 0) AND (upper_nibble <= 9)) then
            upper_nibble = upper_nibble + 48
    else
        ' The value is between 10 (a) and 15 (f).

        upper_nibble = upper_nibble + 87
    endif

    lower_nibble = (value_to_convert & $0f)

    if ((lower_nibble >= 0) AND (lower_nibble <= 9)) then
            lower_nibble = lower_nibble + 48
    else
        ' The value is between 10 (a) and 15 (f).

        lower_nibble = lower_nibble + 87
    endif

return
```

C18
```
void byte_to_ascii_hex(unsigned char value_to_convert, char converted_value[])
{
    char upper_nibble;
    char lower_nibble;

    // Get each hex digit and convert it to a character code.

    upper_nibble = (value_to_convert & 0xf0) >> 4;

    if ((upper_nibble >= 0) && (upper_nibble <= 9))
        upper_nibble += 48;
    else
    {
        // The value is between 10 (a) and 15 (f).

        upper_nibble += 87;
    }
    lower_nibble = value_to_convert & 0x0f;

    if ((lower_nibble >=0) && (lower_nibble <= 9))
        lower_nibble += 48;
    else
    {
        // The value is between 10 (a) and 15 (f).

        lower_nibble += 87;
    }
    converted_value[0] = upper_nibble;
    converted_value[1] = lower_nibble;
}
```

The ascii_hex_to_byte function converts two ASCII hex bytes (upper_nibble and lower_nibble) to the binary value they represent:

PBP
```
ascii_hex_to_byte:

    ' Set converted_byte to the value represented by ASCII Hex
    ' characters upper_nibble and lower_nibble
    ' Set success = 1 on success, 0 on failure.

    success = 1
```

```
        ' Convert each character code to the value it represents.

        select case upper_nibble

            case "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
                upper_nibble = upper_nibble - 48

            case "a", "b", "c", "d", "e", "f"
                upper_nibble = upper_nibble - 87

            case "A", "B", "C", "D", "E", "F"
                upper_nibble = upper_nibble - 55

            case else
                ' The text character isn't 0-9, a-f, or A-F.

                success = 0
        end select

        select case lower_nibble

            case "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
                lower_nibble = lower_nibble - 48

            case "a", "b", "c", "d", "e", "f"
                lower_nibble = lower_nibble - 87

            case "A", "B", "C", "D", "E", "F"
                lower_nibble = lower_nibble - 55

            case else
                ' The text character isn't 0-9, a-f, or A-F.

                success = 0
        end select

        if (success = 1) then

            ' Combine the nibbles in a byte.
            converted_byte = (upper_nibble << 4) + lower_nibble
        endif
return
```

C18
```
int ascii_hex_to_byte(char upper_nibble, char lower_nibble)
{
        // Return the byte value represented by the ASCII Hex
        // characters upper_nibble and lower_nibble
        // Return -1 on failure.

        byte return_value = 0;

        // Convert each char into its value.

        switch (upper_nibble)
        {
                case '0': case '1': case '2': case '3': case '4':
                case '5': case '6': case '7': case '8': case '9':
        {
                upper_nibble -= 48;
                break;
        }
        case 'a': case 'b': case 'c': case 'd': case 'e': case 'f':
        {
                upper_nibble -= 87;
                break;
        }
        case 'A': case 'B': case 'C': case 'D': case 'E': case 'F':
        {
                upper_nibble -= 55;
                break;
        }
        default:
        {
                return_value = -1;
                break;
        }
}
```

```
    switch (lower_nibble)
    {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
        {
            lower_nibble -= 48;
            break;
        }
        case 'a': case 'b': case 'c': case 'd': case 'e': case 'f':
        {
            lower_nibble -= 87;
            break;
        }
        case 'A': case 'B': case 'C': case 'D': case 'E': case 'F':
        {
            lower_nibble -= 55;
            break;
        }
        default:
        {
            lower_nibble = -1;
            break;
        }
    }
    if ((upper_nibble > -1) && (lower_nibble > -1))
        return_value = (upper_nibble << 4) + lower_nibble;
    else
        return_value = -1;

    return return_value;
}
```

Decoding Received Data

Firmware must decode received data to find out if it contains a supported command.

Examining Received Bytes

On receiving a byte and storing it in the char variable serial_in, a node can examine and process the byte:

PBP

```
receive_serial_data:

    ' Process received bytes.

    if (PIR1.5 = 1) then

        ' A byte is available to read.

        if (RCSTA.2 = 1) then

            ' Framing error. Read RCREG to clear the error
            ' but don't use the data.

            hserin [serial_in]

        else
            ' A character was received without a framing error.

            hserin [serial_in]

            select case serial_in

                case $0a

                    ' A LF character was received,
                    ' indicating the end of a command.

                    gosub respond_to_command

                case $0d

                    ' Ignore a received CR character.
```

```
                        case COMMAND_START

                                ' A new command has begun.
                                ' Initialize the array that holds received bytes.

                                received_command[0] = COMMAND_START
                                command_index = 1

                        case else

                                ' A character was received and it's not a LF or CR.
                                ' If at the end of the array, ignore additional received data.

                                if (command_index <= MAX_COMMAND_LENGTH) then

                                        ' Convert characters A-Z to lower case.

                                        if ((serial_in >= "A") and (serial_in <= "Z")) then
                                                serial_in = serial_in + 32
                                        endif

                                ' Save the character and increment the position
                                ' in the array that stores received text.

                                received_command[command_index] = serial_in
                                command_index = command_index + 1
                                endif
                        end select
                endif
        endif
return
```

C18
```
void receive_serial_data(void)
{
    // Process received bytes.

    if (DataRdyUSART())
    {
        // The serial port has received a byte.
        // Read a received character.

        if (RCSTAbits.OERR == 1)
        {
            // Overrun error.
            // Clear the error and re-enable the receiver.

            RCSTAbits.CREN = 0;
            RCSTAbits.CREN = 1;
        }
        if (RCSTAbits.FERR == 1)
        {
            // Framing error.
            // Reading a byte clears the error.
            // Read the byte but don't use it.

            serial_in = getcUSART();
        }
        else
        {
            // A character was received without a framing error.

            serial_in = getcUSART();

            switch (serial_in)
            {
                case 0x0a:
                {
                    // A LF character was received, indicating the end
                    // of a command.

                    respond_to_command();

                    break;
                }
```

```
                    case 0x0d:
                    {
                        // Ignore a received CR character.

                        break;
                    }

                    case COMMAND_START:
                    {
                        // A new command has begun.
                        // Initialize the array that holds received characters.

                        received_command[0]=COMMAND_START;
                        command_index = 1;
                        break;
                    }
                    default:
                    {
                        // A character was received and it's not a LF or CR.
                        // Save the character and increment the position in
                        // the variable that stores received text.
                        // Convert the char to lower case to make the input
                        // case insensitive.

                        // If at the end of the array,
                        // ignore additional received data.

                        if (command_index <= MAX_COMMAND_LENGTH)
                        {
                            received_command[command_index] =
                                tolower(serial_in);
                            command_index++;
                        }
                        break;
                    }
                }
            }
        }
    }
```

Responding to Commands

The respond_to_command routine responds to the two commands defined earlier in this chapter. The function executes on detecting a received LF.

PBP

```
respond_to_command:

        ' On receiving a valid command, call a routine to handle the specific command.

        ' Every command begins with a COMMAND_START code, the node's address,
        ' and a command code.

        if (received_command[0] = COMMAND_START)  then

                if (received_command[1] = MY_ADDRESS) then

                        select case received_command[2]

                                case "1"

                                        gosub command_write_byte

                                case "2"

                                        gosub command_read_byte

                                ‘ Add additonal supported commands here.

                                case else

                                        gosub initialize_serial_buffers

                        end select
                else

                        gosub initialize_serial_buffers

                endif
        else
                gosub initialize_serial_buffers
        endif
return
```

C18

```
void respond_to_command(void)
{
	// On receiving a valid command, call a routine to handle the specific command.
	// Every command begins with a COMMAND_START code, the node's address,
	// and a command code.

	int received_data;

	if (received_command[0] == COMMAND_START)
	{
		if (received_command[1] == MY_ADDRESS)
		{
			// Respond to supported commands.

			switch (received_command[2])
			{
				case '1':
				{
					command_write_byte();
					break;
				}
				case '2':
				{
					command_read_byte();
					break;
				}
				// Add additional supported commands here.

				default:
				{
					initialize_serial_buffers;
					break;
				}
			}
		}
		else
		{
			initialize_serial_buffers;
		}
	}
```

```
        else
        {
                initialize_serial_buffers;
        }
}
```

On receiving an invalid command, the initialize_serial_buffers routine re-initializes the buffers that hold the command and response and thus prepares to receive a new command.

PBP

```
initialize_serial_buffers:

        command_index = 0
        response_index = 0
        received_command[0]= 0
return
```

C18

```
void initialize_serial_buffers(void)
{
        command_index = 0;
        response_index = 0;
        received_command[0]= '\0';
}
```

Writing a Byte

The routine below handles the write_byte command. The routine sets a port bit to match bit 0 of the received byte and prepares to send a response to the primary node. For testing, the port bit can control an LED. A real-world application can use the received data in any way it wants.

PBP

```
command_write_byte:

    ' A write_byte command has been received.

    select case received_command[3]

        case "b"

            ' Get the data to write.
            ' Convert the received ASCII Hex bytes to a byte value.

            upper_nibble = received_command[4]
            lower_nibble = received_command[5]

            gosub ascii_hex_to_byte

            ' Set bit 0 of PortB to match bit 0 in the received byte.

            if ((converted_byte & 1) = 1) then
                high PORTB.0
            else
                low PORTB.0
            endif

            gosub prepare_response

        ' Add more cases as needed.

        case else
    end select
return
```

C18
```
void command_write_byte(void)
{
    int received_data;

    switch (received_command[3])
    {
        case 'b':
        {
            // Get the data to write.

            received_data = ascii_hex_to_byte(received_command[4],
                received_command[5]);

            if (received_data > -1)
            {
                // The received data was valid.
                // Set a port bit to match bit 0 of the received byte.

                if ((received_data & 1) == 1)
                {
                    PORTBbits.RB0 = 1;
                }
                else
                {
                    PORTBbits.RB0 = 0;
                }

                command_response[2] = 0x0a;
                prepare_to_respond();
                break;
            }
        }
        // Add more cases as needed.

        default:
        {
            break;
        }
    }
}
```

Reading a Byte

The function below handles the read_byte command. The function reads a port and prepares to send its value in a response to the primary node.

PBP

```
command_read_byte:

    ' A read_byte command has been received.

    select case received_command[3]

        case "b"

            ' Read Port B.

            value_to_convert = PORTB

            ' Convert the value read to ASCII Hex

            gosub byte_to_ascii_hex

            ' Prepare to send the ASCII Hex characters in a response.

            command_response[2] = upper_nibble
            command_response[3] = lower_nibble
            command_response[4] = $0a

            gosub prepare_to_respond

        ' Add more cases as needed.

        case else
    end select
return
```

C18
```
void command_read_byte(void)
{
    switch (received_command[3])
    {
        case 'b':
        {
            // Read Port B.
            // Store the value read and a LF to send in the response.

            byte_to_ascii_hex(PORTB, &command_response[2]);
            command_response[4] = 0x0a;

            prepare_to_respond();
            break;
        }
        // Add more cases as needed.

        default:
            break;
    }
}
```

After the prepare_to_respond routine executes and any required delay elapses, the network is in the "t" state. The task loop calls the transmit_serial_data routine to send the characters in the command_response array to the primary node.

PBP

```
transmit_serial_data:

        ' If a byte is waiting to transmit, send it.

        if firmware_driver_enable then
                driver_enable = 1
        endif

        ' Wait for the transmit shift register to empty.

        while (TXSTA.1 = 0)
        wend

        hserout[command_response[response_index]]

        if (command_response[response_index] = $0a) then

                ' The entire response has been sent.

                if firmware_driver_enable then
                        while (TXSTA.1 = 0)
                        wend
                        driver_enable = 0
                endif

        ' Prepare to receive another command.

        gosub initialize_serial_buffers
        network_state + "r"

        else
                ' Prepare to send the next byte.

                response_index = response_index + 1
        endif
return
```

C18
```
void transmit_serial_data(void)
{
        #if defined firmware_driver_enable
                driver_enable = 1;
        #endif

        while(BusyUSART());
        putcUSART (command_response[response_index]);

        if (command_response[response_index] == 0x0a)
        {
                // The entire response has been sent.

                #if defined(firmware_driver_enable)

                        while(BusyUSART());
                        driver_enable = 0;

                #endif

                // Prepare to receive another command.

                initialize_serial_buffers;
                network_state = 'r';
        }
        else
        {
                // Prepare to send the next byte.

                response_index++;
        }
}
```

14

Inside USB

USB is a hardware interface with defined protocols that enable a host computer to communicate with a variety of peripheral devices. As Chapter 1 explained, PC software can access some USB devices as virtual COM ports (VCPs). This chapter introduces USB and how it relates to virtual COM-port devices.

The USB 2.0 specification is the main document that defines the interface. USB specifications are available from the USB Implementers Forum (USB-IF) (*www.usb.org*). For more a more detailed discussion of USB protocols, see the specification or my book, *USB Complete: Everything You Need to Develop Custom USB Peripherals*.

Hosts and Devices

An RS-232 interface assumes nothing about the contents of the data being transferred. The computer at each end of the cable can use any method to define what the data means.

In contrast, USB is an intelligent interface with defined protocols. Every USB communication is between a host and a device. The host manages communications on the bus, and devices respond to communications from the host. The

host is a PC or another computer that contains host-controller hardware and software. A device is a peripheral or other computer-controlled system that contains device-controller hardware and firmware.

The host assigns a driver or a series of drivers to each attached device. The drivers manage communications between applications and the USB host controller's driver and define how applications can access a device. Drivers can insulate applications from the details of a device's hardware interface. RS-232 and USB are very different interfaces, but with driver support, application software can use the same COM-port functions to access internal RS-232 ports and USB devices that function as virtual COM ports.

On power up or device attachment, the host computer and the device exchange information in a process called enumeration. The host requests a series of data structures called descriptors from the device. The descriptors contain information that identify the device and its intended use. The information includes the device's Vendor ID, Product ID, Release Number and codes that identify any standard USB classes the device belongs to.

During enumeration, the host also assigns a bus address to the device and requests the device to use a configuration that specifies how much current the device can draw from the bus.

Assigning a Driver on the Host

For each device, the host computer looks for the best match between the information in the device's descriptors and the information in the host's INF files (or an equivalent information source for non-Windows systems). The information can be a Vendor ID/Product ID pair or USB class and subclass codes. On finding a match, the host uses the driver or drivers named in the INF file. After the device accepts the requested configuration, the host and device can begin communicating using the assigned drivers.

A virtual COM-port device can use the *serial.sys* and *usbser.sys* drivers provided with Windows or an alternate driver obtained from a device vendor or other source. Some companies provide free COM-port drivers for use with the company's USB controllers.

Requirements

A USB host can be a desktop or notebook computer, a handheld, or an embedded system. The host must contain host-controller hardware and must imple-

ment USB protocols. To communicate with COM-port devices, the host must support the COM-port software interface and must either support the USB communication devices class or use a vendor-specific driver that implements a virtual COM port.

A USB device can contain a microcontroller with an on-chip USB device controller, or the USB device controller can be on a separate chip that interfaces to a microcontroller or other CPU. The device can use a USB controller designed specifically for COM-port applications or a generic controller that can be programmed for any use. The hardware that implements the low-level USB protocols in the device controller is called the serial interface engine (SIE). Some devices manage USB communications entirely in hardware and require no firmware programming to support USB-specific protocols.

An On-The-Go (OTG) device is a USB device that can function as a limited-capability host and as a device (but not both at the same time). An OTG device can function as a host to virtual COM-port devices or as a virtual COM-port device.

Host Responsibilities

A USB host manages power and communications on the bus and has these responsibilities:

- Detect and enumerate all attached devices,.
- Detect when a device has been removed from the bus.
- Provide power to devices and work with the devices to conserve power when possible.
- Manage data flow on the bus.
- Perform error checking.

Device Responsibilities

In many ways, a device's responsibilities mirror the host's, but devices also have unique duties. A USB device has these responsibilities:

- Detect voltage on the bus's power-supply line and on detecting the voltage, switch in a pull-up resistor to announce the device's presence to the host.
- Manage power. In normal operation, a device must limit the bus current consumed to either 100 mA or a higher amount, up to 500 mA, in a configuration supported by the device and requested by the host during enumeration. A device must also detect the presence of the host's periodic timing markers and enter the low-power Suspend state when the markers are absent. While in the

Suspend state, the device must limit its current consumption and monitor the bus, exiting the Suspend state when bus activity resumes.

- Respond to requests sent by the host during and after enumeration.
- Perform error checking.
- Exchange data with the host. A virtual COM-port device receives COM-port data from the host and sends COM-port data as needed to the host.
- As needed, send and receive COM-port parameters and status and control information.

Speed

The USB 2.0 specification defines three bus speeds: high speed at 480 Mbps, full speed at 12 Mbps, and low speed at 1.5 Mbps. USB devices in the communication devices class must support full speed, high speed, or both. Almost all high-speed devices also support full speed because adding support for full speed is rarely difficult and enables the device to work when attached to full-speed hosts. USB hosts in recent PCs support all three speeds.

The bus speeds describe the rate that information travels on the bus. The theoretical maximum data-transfer rate for the bulk endpoints used on most virtual COM ports is 1.216 Megabytes/s at full speed and 53.248 Megabytes/s at high speed. The real-world maximum throughput is less and varies with the programming on the host and device, the hardware capabilities of the host and device, and how busy the bus is.

Endpoints

All bus traffic travels to or from device endpoints. An endpoint serves as a buffer for received data or data waiting to transmit. Typically an endpoint is a block of data memory or a register in the device controller.

Every device must implement endpoint zero, which is bidirectional. A device can have up to 30 additional endpoint addresses. Each of these endpoint addresses has a number (1 to 15) and direction (IN or OUT). The direction is defined from the host's perspective: an IN endpoint provides data to send to the host and an OUT endpoint stores data received from the host. Device hardware or firmware configures each endpoint address for a specific USB transfer type and direction. The number of available endpoints varies with the device controller. A USB virtual COM port typically uses three endpoint addresses in addition to endpoint zero.

Each endpoint address except endpoint zero has an endpoint descriptor. The descriptor's wMaxPacketSize field specifies how many bytes the endpoint can transfer in a data packet. For bulk endpoints, wMaxPacketSize can be up to 64 for full-speed endpoints and must be 512 for high-speed endpoints. (The device descriptor specifies the maximum packet size for endpoint zero.)

USB Transfers

The USB specification defines structures for transferring data on the bus and ensuring that transmitted data reaches its receiver.

Transfer Types

One reason why USB is suitable for a wide range of devices is its support for four types of data transfers (Table 14-1).

Control transfers enable the host to learn about a device, set a device's address, and select configurations and other settings. The host uses control transfers to learn about the device during enumeration. Control transfers can also send class-specific and vendor-specific requests that transfer data for any purpose. A USB host typically uses control transfers to send serial-port parameters for the device to implement. Control transfers can also request to set serial-port control signals. All USB devices must support control transfers. Every device must support control transfers on endpoint zero.

A control transfer has two or three stages. In the Setup stage, the host sends a request and related information in defined fields. The bmRequestType field specifies the direction of data flow, the type of request, and the recipient. The bRequest field identifies the request. The wValue and wIndex fields can contain information specific to the request. The wLength field indicates the number of bytes in the Data state that follows. In the Data stage, the host or device sends data. Some requests don't have a Data stage. In the Status stage, the receiver of data in the Data stage returns status information. If there is no Data stage, the device returns the status information.

The other transfer types don't have stages. A class specification or vendor-specific protocol determines the length of a transfer. Bulk transfers are intended for situations where the rate of transfer isn't critical. If the bus is very busy, bulk transfers must wait, but if the bus is otherwise idle, bulk transfers are the fastest of all. USB virtual COM-port devices typically use bulk transfers for COM-port data. Interrupt transfers are for devices that must receive or send

Table 14-1: Each of USB's four transfer types suit different uses.

Transfer Type	Feature	Use in Virtual COM Ports
Control	Three stages (Setup, Data, Status).	Enumeration. Set and get serial-port parameters. Set serial-port control signals.
Interrupt	Guaranteed maximum latency.	Status information.
Bulk	Fastest on an otherwise idle bus.	COM-port data.
Isochronous	Guaranteed transfer rate but no error detecting.	COM-port data (requires driver support from vendor).

data periodically. USB virtual COM-port devices use interrupt transfers to send status information to the host. Isochronous transfers have guaranteed delivery time but no error correcting. Real-time audio uses isochronous transfers. With driver support, USB virtual COM-port devices can use isochronous transfers for COM-port data in place of bulk transfers.

Transactions

Each USB transfer consists of one or more transactions (Table 14-2). Each transaction contains a token packet, a data packet, and a handshake packet. (An exception is isochronous transfers, which don't have handshake packets.) Each packet begins with a packet ID (PID). The function of the PID varies with the packet type.

The token packet contains the device address and the endpoint number the transaction is directed to. The token packet's PID identifies the type of packet: SETUP (the first packet in a control transfer), OUT (other host-to-device packet), IN (device-to-host packet), or SOF (start-of-frame marker).

The data packet contains any data the host or device is sending in the transaction. For control transfers, the transfer's stage and the specific request determine who sends the data. For other transfers, the endpoint's direction determines who sends the data. The PID contains the data-toggle value, explained below.

The receiver of the data packet (or the device if there is no data packet) sends the handshake packet. The PID contains a code to indicate the status of the transaction. ACK means success. NAK on an IN transaction means the device has no data to send. NAK on an OUT transaction means the device was too busy to accept the data sent. (The host can try again later.) STALL means the device doesn't support a received request in a control transfer or the endpoint's

Table 14-2: Transactions that carry control-transfer requests and other data contain token, data, and handshake packets.

Packet Type	Contents	Source
Token	Device address, endpoint number and direction, transaction type	Host
Data	Transaction-specific data. Not used in some transactions.	Host or device
Handshake	Status code	In transactions with a data packet, the receiver of the data packet. In transactions with no data packet, the device. Not used in isochronous transactions.

Halt feature is set. High-speed bulk OUT endpoints can also return a NYET handshake code, which means that the endpoint accepted the data in the current transaction but isn't yet ready for more data.

The Data Toggle

The data toggle is a data-sequencing value that guards against lost or duplicated data. If you're debugging a device where it appears that the proper data is transmitting on the bus but the receiver is discarding the data, chances are good that the device isn't sending or expecting the correct data toggle.

Each endpoint address maintains its own data-toggle value, which alternates between DATA0 and DATA1. Devices typically store the value in a register bit. When the host configures a device on power up or attachment, the host and device each set their data toggles to DATA0. On detecting an incoming data packet, the host or device compares the state of its data toggle with the data toggle in the received data packet. If the values match, the data packet's receiver toggles its value for the next transaction and returns an ACK. On receiving the ACK, the data packet's sender toggles its value for the next transaction.

The next received packet should contain a data toggle of DATA1, and again the receiver toggles its bit and returns an ACK. In additional transactions, the data toggle continues to alternate between DATA0 and DATA1. An exception is control transfers, where the Status stage always uses DATA1.

If the receiver is busy and returns a NAK, or if the receiver detects corrupted data and returns no response, the sender doesn't toggle its bit and tries again with the same data and data toggle.

Control transfers always use DATA0 in the Setup stage, use DATA1 in the first transaction of the Data stage, toggle the value in any additional Data-stage transactions, and use DATA1 in the Status stage. Bulk and interrupt endpoints toggle the value in every transaction, resetting the data toggle only after a bus reset or on completing a Set Configuration, Set Interface, or Clear Feature(ENDPOINT HALT) request.

15

Using Special-function USB Controllers

Vendors who offer USB controllers designed specifically for use in virtual COM ports are Future Technology Devices International (FTDI) (*www.ftdichip.com*), Prolific Technology, Inc. (*www.prolific.com.tw*), and Silicon Laboratories (*www.silabs.com*). The chips from these vendors function in similar ways. These chips are sometimes called USB to UART bridges. This chapter describes FTDI's chips and driver.

FTDI has several USB interface chips that manage enumeration and other bus communications completely in hardware. The company provides free USB virtual COM-port drivers for use with the chips. The chips don't require any USB-specific firmware at all but can store device-specific descriptor values in EEPROM if desired. A microcontroller or other CPU interfaces to the controller via a serial or parallel interface. The CPU provides COM-port data for the controller to send to the USB host and retrieves COM-port data received from the USB host.

Inside the Chips

The FT232R USB UART and FT245R USB FIFO are USB controllers that can each appear as a virtual COM port on a USB host. Each chip has a full-speed USB device port. The FT232R converts between USB and an asynchronous serial interface, and the FT245R converts between USB and a parallel interface. Both chips have a 128-byte transmit buffer and a 256-byte receive buffer. The chips use bulk transfers to send and receive COM-port data.

Another controller from FTDI is the FT2232C Dual USB UART/FIFO. The chip contains two controllers that each support several configurations, including two virtual COM ports in one chip.

Serial Interface (FT232R)

Figure 15-1 shows the pin functions for the FT232R USB UART. The USBDM and USBDP pins interface to the USB data lines. TXD and RXD are the asynchronous serial output and input. Six pins support status and control signals that correspond to the signals defined in the RS-232 standard. TXD, RTS#, and DTR# are outputs, and RXD, CTS#, DSR#, RI#, and DCD# are inputs.

On receiving asynchronous serial data at RXD, the chip sends the data to the USB host in bulk transfers. On receiving data in bulk transfers from the USB host, the chip writes the data in asynchronous serial format to the TXD pin. The asynchronous serial port can transfer up to 300 kbytes/s of data assuming one Start bit and one Stop bit per byte.

Five additional pins form the configurable CBUS. The pins are programmable via a utility provided by FTDI and can have these functions:

- LED driver that pulses when transmitting or receiving via USB.
- Output that goes low when the device is in the USB Suspend state.
- General-purpose I/O.
- Output that goes high when data is transmitting on TXD. The output can interface directly to the transmit-enable input of an RS-485 transceiver, eliminating the need to enable the transmitter using firmware or additional hardware.
- Configurable clock output for driving an external CPU or microcontroller.

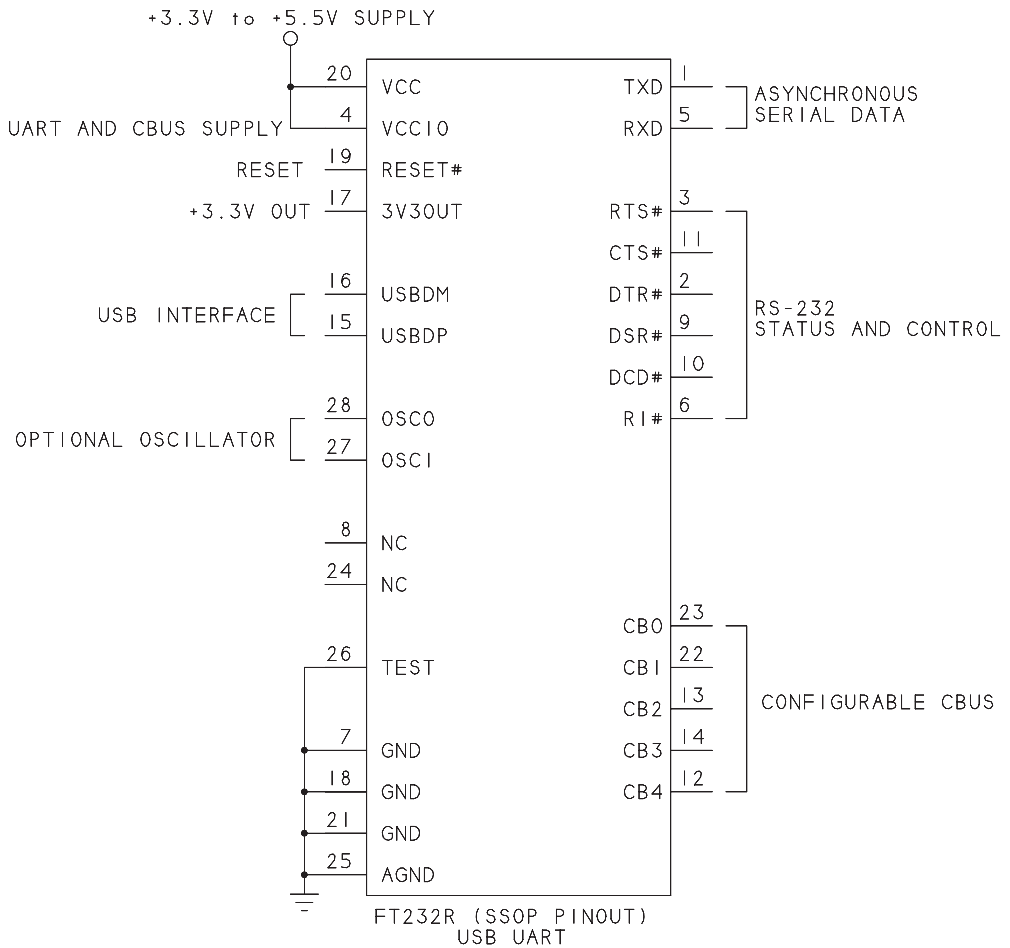


Figure 15-1: The FT232R converts between USB and asynchronous serial data.

The chip can obtain its power from the VBUS line in the USB cable or from an external supply that connects to the VCC pin. The 3V3OUT pin is a +3.3V output. VCCIO is an input whose voltage determines the logic levels on the UART and CBUS pins. The pin can connect to VCC, 3V3OUT, or an external supply. The chip doesn't require an external oscillator.

To convert the asynchronous serial interface to RS-232, use a Maxim MAX232 or similar interface chip. To convert to RS-485, use an SN75176B or similar RS-485 transceiver.

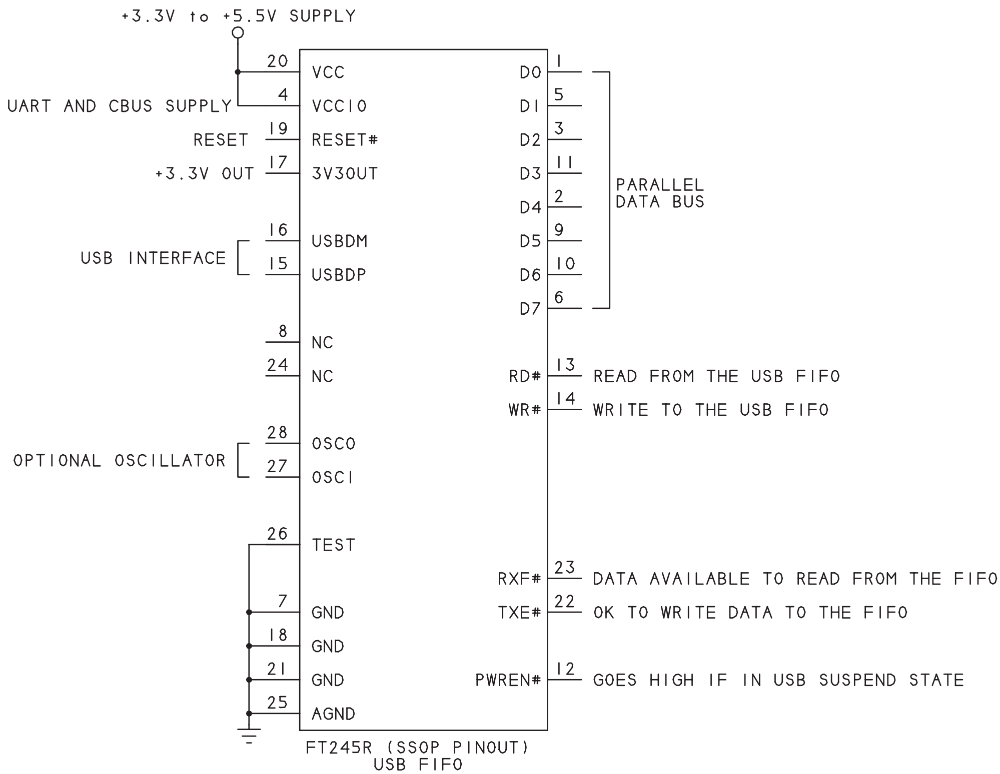


Figure 15-2: The FT245R converts between USB and parallel data.

Parallel Interface (FT245R)

Figure 15-2 shows the pin functions for the FT245R USB FIFO. The USB interface and power options are the same as for the FT232R. But instead of an asynchronous serial port, the chip has an 8-bit, bidirectional parallel data bus with status and control pins that control reading and writing to the buffer. (A FIFO is a buffer whose contents are read in the same order as they were stored. In other words, the first byte written to the buffer is the first byte read from the buffer.) On receiving data on the parallel port, the chip sends the data to the USB host. On receiving data from the USB host, the chip makes the data available to read at the parallel port.

The status and control signals are named from the perspective of the external CPU or microcontroller that interfaces to the chip. The RXF# output is low

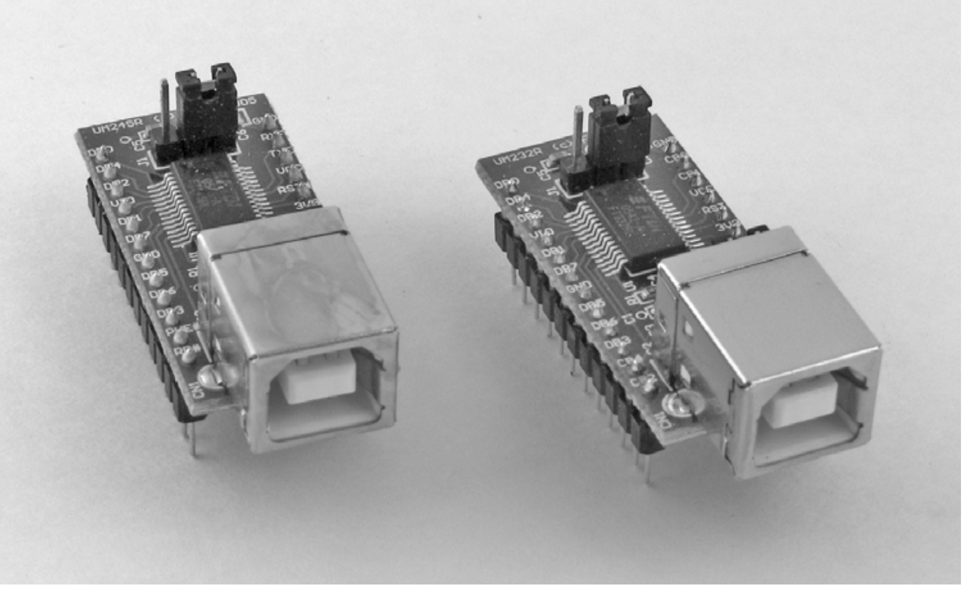


Figure 15-3: For easy prototyping with FTDIs' controllers, use the UM232R and UM245R modules.

when the CPU can read a byte from the FT245R, and the CPU strobes RD# to read the byte. In the other direction, the TXE# output goes low when the CPU can write a byte to the FT245R, and the CPU strobes WR# to write the byte into the FT245R's buffer. An external CPU can use a data bus or any spare port pins to access the FT245R's parallel port. Parameters such as bit rate and parity don't apply to this chip.

Prototyping Modules

FTDI's controllers are available only in surface-mount packages. For prototyping, FTDI and other vendors offer a variety of converters and development boards. FTDI's UM232R and UM245R modules (Figure 15-3) each consist of a circuit board with a controller chip, USB connector, and related circuits mounted on a 24-pin dual in-line package (DIP).

For interfacing a PC to a microcontroller's serial port, FTDI's TTL-232R serial converter cable provides a quick solution. Inside the cable's USB connector is an FT232R circuit. The other end of the cable has a 6-pin socket header that

provides access to the pins most often used by microcontroller serial interfaces: data (TX, RX) and flow control (RTS#, CTS#) plus power and ground (VCC, and GND.

DLP Design (*www.dlpdesign.com*) offers modules that add PIC microcontrollers, sensors, and other useful components interfaced to FTDI's controllers.

Using the Controllers

Unlike other USB controllers, the FT232R and FT245R aren't designed as general-purpose devices that can be programmed to use any host driver. Instead, the devices are intended for use with drivers provided by FTDI.

Drivers

Using the driver provided by FTDI, applications can access a chip as a virtual COM port or via a vendor-specific API.

In most cases, using an FT232R to convert an RS-232 device to USB requires no changes to application software. Applications can continue to access the device as a COM port. An FT245R functions as a virtual COM Port that transfers parallel data.

For applications that don't want to use COM-port functions, the driver's D2XX DLL defines a vendor-specific API. Both the FT232R and FT245R can use this API to access the devices. FTDI provides INF files that enable Windows to match the driver to a device.

Adding Vendor-specific Data

Both controllers contain on-chip EEPROM that can store vendor-specific values for items such as a Vendor ID, Product ID, serial-number string, other descriptive strings, and values that specify whether the device is bus- or self-powered. If there is no EEPROM data for an item, the controller uses a default value. FTDI provides a utility that programs the information into the EEPROM. Older versions of the controllers interface to external EEPROMs.

By default, the chips use FTDIs' Vendor ID and Product ID. On request, FTDI will grant the right for your device to use their Vendor ID with a Product ID that FTDI assigns to you. Of course you can use your own Vendor ID and Product ID if you wish.

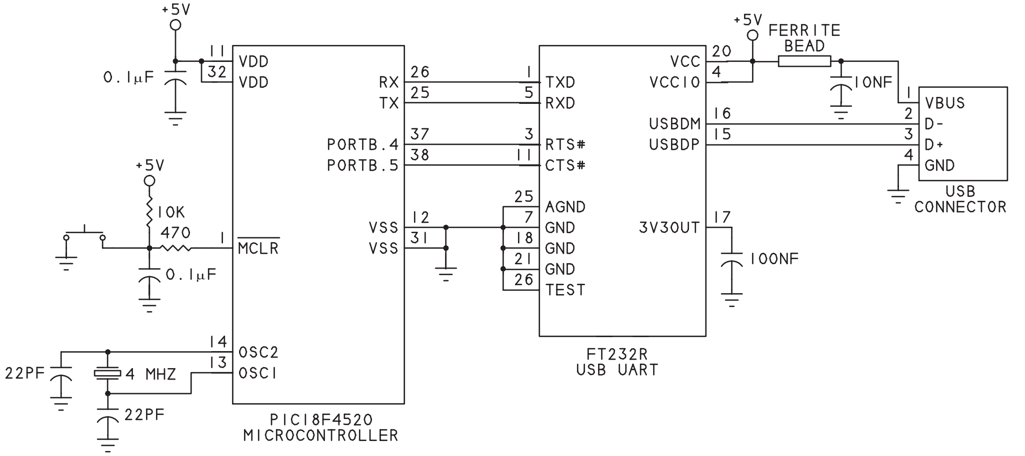


Figure 15-4: This USB virtual COM port uses an asynchronous serial interface to a PIC microcontroller.

Implementing a Virtual COM Port

Figure 15-4 shows a circuit that can serve as a virtual COM port. The USB controller is an FT232R, which interfaces to a PIC18F4520 microcontroller. The PIC18F4520's asynchronous serial output (TX) connects to the FT232R's asynchronous serial input (RXD), and the FT232R's asynchronous serial output (TXD) connects to the PIC18F4520's asynchronous serial input (RX).

The example circuit uses the RTS# and CTS# handshaking lines on the FT232R. These lines can connect to any otherwise unused port pins on the PIC18F4520. If needed, the FT232R's other status and control pins can connect to other port pins on the PIC18F4520.

Attach the USB connector to a port on a PC and follow the on-screen instructions to install FTDI's drivers. (See FTDI's website for a detailed driver-installation guide.) When the drivers are installed, the Device Manager shows a new COM port, and the port is ready for use.

In a similar way, any microcontroller with an asynchronous serial port (and additional port pins if needed) can interface to an FT232R and function as a USB virtual COM port.

Converting from RS-232 to USB

If you have an existing design that uses RS-232, an FT232R chip can quickly convert the design to USB. Figure 15-5 shows a circuit that uses RS-232 along with the same circuit converted to use USB.

In the RS-232 circuit, microcontroller pins connect to a MAX237 interface chip that converts between TTL and RS-232 logic levels. The example circuit includes the asynchronous serial interface (TX and RX) plus six generic I/O pins to support RS-232's status and control signals. A circuit that doesn't use all of the status and control signals requires fewer drivers and receivers. The schematic doesn't show the MAX237's capacitors or power and ground connections.

In the USB circuit, the microcontroller port pins connect to the corresponding pins on an FT232R USB UART. Instead of an RS-232 connector, the circuit has a USB connector that interfaces to the FT232R. The pins for any unimplemented signals can remain open on the FT232R or can be tied to a desired state.

In most cases, a circuit modified in this way requires no changes to device firmware or applications that communicate with the device. Compared to a an internal RS-232 port, a USB virtual COM port can have these differences at the host:

- Longer delays when reading or writing to RS-232 status and control signals.
- Longer delays when changing the parity type (such as in 9-bit networking).

In addition, data throughput at the receiving application can be slow in these situations:

- Receiving less than 62 data bytes. The device prefers sending data packets of wMaxPacketSize with each packet consisting of 62 data byes plus two status bytes. A device that has less than 62 data bytes to send will wait for one of the following events to be true: the chip has accumulated 62 bytes to send, or the chip's internal latency timer has timed out. The default latency time is 16 ms.
- Receiving large blocks of data at bit rates of 38.75 kbps or higher. When the device continuously provides data packets of wMaxPacketSize, the driver holds the received data and passes it to the application when one of the following is true: the driver has received 4096 bytes, or the latency timer has timed out. At fast bit rates, the result can be delayed data at the application.

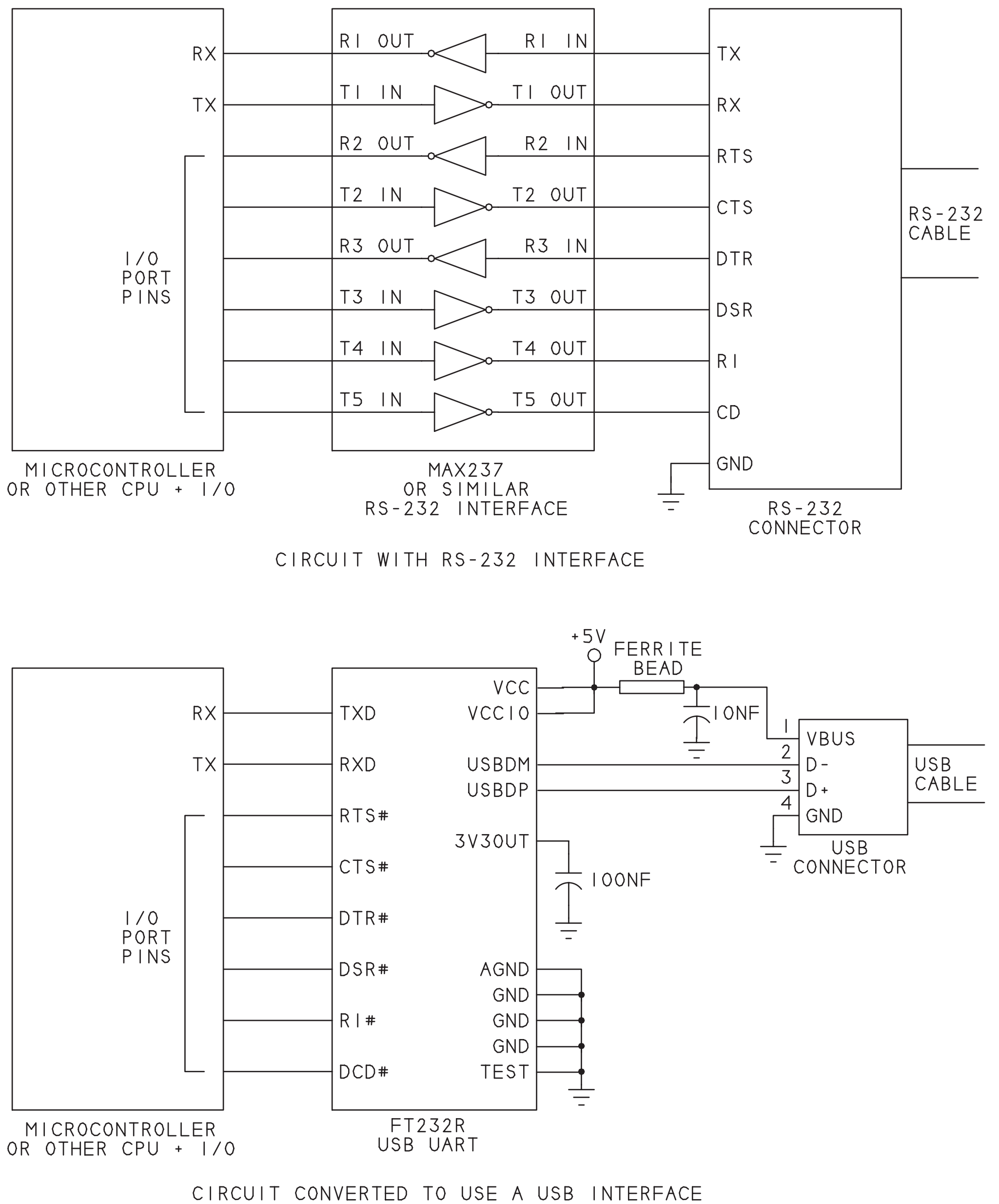


Figure 15-5: To convert a circuit from RS-232 to USB, replace the connections to RS-232 interface chip(s) with connections to an FT232R USB UART.

For techniques to improve data throughput in the above situations, see FTDI's application note *AN232B-04: Data Throughput, Latency, and Handshaking*.

16

Using Generic USB Controllers

Chapter 15 described how to use USB device controllers designed specifically for virtual COM-port applications. USB virtual COM-port devices can also use just about any generic full- or high-speed USB device controller with supporting firmware. As with any USB device, the descriptors returned by the device determine which INF file the PC uses to identify drivers for the device. A USB virtual COM-port device can use drivers provided by Windows or vendor-specific drivers.

USB virtual COM-port devices that use the drivers provided by Windows belong to the USB communication devices class. This chapter introduces the class and its use in virtual COM-port devices.

The Communication Devices Class

The USB communication devices class (CDC) encompasses a variety of devices, including telephones and "medium-speed" networking devices. The telephones group includes generic virtual COM port devices as well as analog

phones and modems, ISDN terminal adapters, and cell phones. Networking devices include ADSL modems, cable modems, and Ethernet adapters and hubs. The USB interface in these devices can carry data that uses application-specific protocols such as V.25ter/V.250 for modem control or Ethernet for a network.

Documentation

The CDC specifications are available from the USB-IF (*www.usb.org*). The class includes a series of subclasses (Table 16-1). The main CDC specification defines most of the subclasses. WMC and EEM were defined after the release of the CDC specification and thus have their own documents. The CDC standard refers to the V.25ter standard, which evolved from the AT command set for modems. A more recent edition of the standard is V.250 (*www.itu.int*). The OBEX protocol used by some WMC devices is defined in *IrDA Object Exchange (OBEX) Protocol* from *www.irda.org*.

Overview

A USB CDC device is responsible for device management, call management if needed, and data transmission. Device management includes controlling and configuring the device and notifying the host of events. Call management includes establishing and terminating telephone calls or other connections. Some devices don't need call management. Data transmission is the sending and receiving of application data such as phone conversations, files, or other data sent over a modem or network.

CDC supports five basic models for communicating. Each model encompasses one or more subclasses.

- The POTS (Plain Old Telephone Service) model is for devices that communicate via ordinary phone lines and generic COM-port devices. Ethernet devices that comply with Microsoft's USB Remote Network Driver Interface Specification (NDIS) also use the POTS model.
- The ISDN model is for communications via phone lines with ISDN interfaces.
- The networking model is for communicating via Ethernet or ATM (asynchronous transfer mode) networks.
- The wireless mobile communications (WMC) model includes cell phones that support voice and data communications.

Table 16-1: The CDC communication class interface supports a variety of subclasses. In an interface descriptor, the bInterfaceSubClass field contains the subclass code.

Model Category	Code (Hex)	SubClass	Use
–	00	Reserved	–
POTS	01	Direct Line Control	Phone modem with the host providing any data compression and error correction. The device or host may modulate and demodulate the modem data.
	02	Abstract Control	Phone modem with the device providing any data compression, error correction, and data modulation and demodulation. Also used for generic virtual COM ports (serial emulation) and Ethernet via Remote NDIS
	03	Telephone Control	Multi-line phone control
ISDN	04	Multi-Channel Control	Multiple channels multiplexed on a network interface
	05	CAPI Control	COMMON-ISDN-API (CAPI) commands and messages.
Networking	06	Ethernet Networking Control	Ethernet-framed data
	07	ATM Networking Control	Asynchronous transfer mode (ATM) functions
WMC	08	Wireless Handset Control	Control of logical handset
	09	Device Management	Management communications between the handset and host
	0A	Mobile Direct Line	Direct control of the mobile terminal radio component
	0B	OBEX	Support for Object Exchange (OBEX) protocol
EEM	0C	Ethernet Emulation	Efficient transfer of Ethernet-framed data
–	0D–7F	Reserved (future use)	–
–	80–FE	Reserved (vendor specific)	–

- The Ethernet emulation model (EEM) defines an efficient way for devices to send and receive Ethernet frames.

Note that there are three options for Ethernet devices: the networking model (Ethernet networking control subclass), the more recently defined Ethernet emulation model, and the POTs model (abstract control subclass) using the Microsoft-specific NDIS.

Notifications announce events such as changes in RS-232 status signals. Devices typically use an interrupt endpoint to send notifications, though the standard allows using bulk endpoints. Each notification contains an 8-byte header with a 2-byte field for data. Some notification types include additional data that follows the header.

Class-specific control requests enable the host to get and set communication parameters and status and control signals.

For reliable data transfers, many cell phones use the OBEX protocol originally developed for infrared communications. The WMC specification defines an OBEX model subclass for this purpose.

Device Controllers

Most virtual COM-port devices use one interrupt endpoint address and two bulk endpoint addresses. The controller must support full or high speed because low-speed USB doesn't allow bulk transfers. Just about any full- or high-speed USB device controller will have the needed number of endpoints for a CDC device.

Host Drivers

The *usbser.sys* driver included with Windows 98 SE and later is suitable for use with modems and virtual COM ports. Each device that uses *usbser.sys* must have an INF file that contains the device's USB Vendor ID and Product ID. Windows doesn't provide a generic INF file for USB virtual COM-port devices as it does for other device classes.

For better performance, some devices use vendor-specific drivers that perform the functions of the *serial.sys, usbser.sys,* and *serenum.sys* drivers included with Windows. Limitations of the Windows drivers in USB virtual COM-port communications include slow performance, lack of support for reading RS-232's CTS status signal at the host, and limited support for composite devices with CDC interfaces. OBEX support requires a vendor-provided driver. New Win-

dows editions and service packs have brought improvements in *usbser.sys,* so using a recent and updated Windows edition is likely to give the best results.

An alternate USB virtual COM-port driver from Walter Oney Software (*www.oneysoft.com*) provides enhancements such as the ability for an application to retain an open handle when a device is removed and re-attached. Other sources for USB virtual COM-port drivers include MCCI (*www.mcci.com*) and Thesycon Systemsoftware & Consulting (*www.thesycon.de*).

For telephone functions, the Windows Telephony Application Programming Interface (TAPI) defines a standard way for applications to control telephone functions for data, fax, and voice calls. TAPI manages signaling, including dialing, answering, and ending calls and supplemental services such as holding, transferring, and conference calls.

Using the Abstract Control Model

The POTS model encompasses several models that vary in the amount of data processing the device is responsible for. Each model supports different requests and notifications. Processing can include modulation and demodulation, error correction, and data compression.

As Chapter 8 explained, modulation is the process of encoding information by varying the amplitude, frequency, phase, or a combination of these on a signal to be transmitted. Demodulation is the reverse process of extracting data from a modulated signal. In a basic wired, digital interface, the data doesn't use a carrier or modulation.

POTS Models

The most basic POTS model is the direct line control model, which in turn encompasses the Direct Line (DL), datapump, abstract control, and telephone control models.

In the DL model, the device converts between the phone line's analog signal and digital data, and the host handles modulation and demodulation, error correction, and data compression. The data uses an audio interface. The datapump model is similar except the device handles modulation and demodulation, and the data uses a vendor-specific interface.

In the abstract control model, the device handles modulation and demodulation and detects and responds to V.25ter commands. The host or device can handle error correction and data compression. The model supports requests

and notifications to get and set RS-232 status and control signals and asynchronous port parameters. Data uses a CDC data interface.

Devices that use the abstract control model include virtual COM ports and Ethernet devices that comply with the Microsoft-specific Remote NDIS protocol. The *Remote NDIS USB Driver Kit*, available from Microsoft, includes Microsoft's specification.

The CDC telephone control model supports call management and notifications and typically uses other classes such as audio for data and HID for a keypad interface.

Virtual COM Ports

The function performed by Virtual COM-port devices is sometimes called serial emulation. The CDC specification says that abstract-control-model devices understand AT commands. In practice, however, a vitual COM-port device that doesn't communicate with a modem that uses the commands will never receive a command and thus doesn't need to support them.

A generic COM-port device that supports the abstract control model and uses the standard Windows drivers performs these tasks:

- Returns descriptors that identify the device as a COM-port device that supports the abstract control model and common AT commands.
- Accepts COM-port data on a bulk OUT endpoint.
- Sends COM-port data as needed on a bulk IN endpoint.
- Sends SERIAL_STATE notifications as needed on an interrupt IN endpoint.

In addition, most abstract control model devices support class-specific requests to get and set asynchronous port parameters and to control the RS-232 signals RTS, DTR, and Break.

The abstract control model requires devices to support the class-specific control requests SEND_ENCAPSULATED_COMMAND and GET_ENCAPSULATED_RESPONSE and the class-specific notification RESPONSE_AVAILABLE. Again, devices that don't use AT commands will never receive these requests or need to send the notification.

The model doesn't define a way for the host to read the state of RS-232's CTS status signal. Device firmware can still read CTS on a local asynchronous port and take appropriate action. For example, if a virtual COM-port device has data to send to a remote device that hasn't asserted CTS, the virtual COM-port device can store the data in a buffer and wait to transmit. If the buffer is full,

the virtual COM-port device can NAK attempts by the USB host to send data. When the remote device asserts CTS, the virtual COM-port device can send the buffered data and begin accepting new data from the host. To use CTS in this way, the USB host doesn't need to know the signal's state.

If you want to use CTS in an unconventional way, such as having a host application read a switch state on a device, you're out of luck unless you can define a vendor-specific command that travels on the same bulk pipes that carry application data or use a vendor-specific driver that supports reading CTS.

Example Firmware

Some chip companies provide example firmware for USB virtual COM ports using generic microcontrollers that contain USB device controllers. Chips with example code include Atmel Corporation's AT89C5131, Microchip Technology's PIC18F4550, and NXP Semiconductors' LPX214x. Example firmware can be extremely helpful in getting a device up and running. Any of these examples is a good supplement to the material in this chapter.

Requests

Table 16-2 shows required and optional requests for devices that use the abstract control model.

The two required requests, SEND_ENCAPSULATED_COMMAND and GET_ENCAPUSULATED_RESPONSE, enable the host to perform protocol-specific communications such as issuing an AT command or requesting a response to a command. If the COM-port device doesn't use AT commands, the host will never send these requests.

For devices that have asynchronous serial interfaces, other requests monitor and control the states of RS-232 signals that these devices might use. The SET_LINE_CODING and GET_LINE_CODING requests set and request

Table 16-2: The abstract control model defines a set of required and optional requests that travel in control transfers.

Request	Code	Description	Required?
SEND_ENCAPSULATED_COMMAND	0x00	Issues a command in the format of the supported control protocol.	Required but not used if AT commands aren't implemented.
GET_ENCAPSULATED_RESPONSE	0x01	Requests a response in the format of the supported control protocol.	Required but not used if AT commands aren't implemented.
SET_COMM_FEATURE	0x02	Controls a communication feature.	Optional.
GET_COMM_FEATURE	0x03	Returns the current settings for a communication feature.	Optional.
CLEAR_COMM_FEATURE	0x04	Clears a communication feature.	Optional.
SET_LINE_CODING	0x20	Sets asynchronous serial parameters: bit rate, number of Stop bits, parity, and number of data bits.	Recommended for devices that support these parameters.
GET_LINE_CODING	0x21	Requests asynchronous serial parameters: bit rate, number of Stop bits, parity, and number of data bits.	Recommended for devices that support these parameters.
SET_CONTROL_LINE_STATE	0x22	Set RS-232 signals RTS and DTR.	Optional.
SEND_BREAK	0x23	Set RS-232 Break signal.	Optional.

serial port parameters. The line-coding information travels in the data stage of the request:

Offset	Field	Size (bytes)	Description
0	dwDTERate	4	Bit rate (bits per second)
4	bCharFormat	1	Stop bits: 0 = 1 Stop bit 1 = 1.5 Stop bits 2 = 2 Stop bits
5	bParityType	1	Parity: 0 = None 1 = Odd 2 = Even 3 = Mark 4 = Space
6	bDataBits	1	Number of data bits (5, 6, 7, 8, or 16)

In the SET_CONTROL_LINE_STATE request, the host tells the device how to set the RS-232 control signals RTS and DTR. Two bits in the request's wValue field contain the information:

wValue Bit	Description
15..2	Reserved. (Set to zero.)
1	RTS: 0: de-assert (RS-232 negative voltage) 1: assert (RS-232 positive voltage)
0	DTR: 0: de-assert (RS-232 negative voltage) 1: assert (RS-232 positive voltage)

The SEND_BREAK request requests the device to send an RS-232 break signal for the number of milliseconds specified in the wValue field of the request. If wValue = FFFFh, the device should maintain the break signal until receiving another SEND_BREAK signal with wValue = 0000h.

The GET_COMM_FEATURE, SET_COMM_FEATURE, and CLEAR_COMM_FEATURE requests can request, set, or clear an ABSTRACT_STATE data structure used in call management functions and a COUNTRY_SETTING code. These requests aren't relevant for generic COM-port devices.

Notifications

Abstract control model devices support up to three notifications:

Notification	Code	Description	Required?
NETWORK_CONNECTION	0x00	Network or modem connection status.	Optional.
RESPONSE_AVAILABLE	0x01	Requests the host to issue a GET_ENCAPSULATED_ RESPONSE request.	Required but not used if AT commands aren't implemented.
SERIAL_STATE	0x20	State of CD, DSR, Break, and RI	Optional but recommended for devices that support these signals.

Each notification begins with an 8-byte header:

Field	Bytes	Description
bmRequestType	1	0x0A
bNotification	1	Notification code
wValue	2	Notification-specific data
wIndex	2	bInterfaceNumber
wLength	2	Number of data bytes to follow (zero or more)

The device typically returns notifications on the interface's interrupt IN endpoint. The host doesn't request specific notifications. Instead, the host periodically issues IN token packets to the endpoint, and the endpoint returns notifications as available. The endpoint descriptor's bInterval field specifies the maximum latency between IN token packets. An endpoint that has no notification to return should return NAK on receiving an IN token packet.

Devices that support the abstract control model must support the RESPONSE_AVAILABLE notification. However, if the device is a generic COM-port device that doesn't connect to a modem that supports AT commands, the device will have no need for this notification and will return NAK on every IN token packet to the notification endpoint.

The SERIAL_STATE notification is optional but recommended for devices with asynchronous serial interfaces. The notification sends the states of RS-232

signals and other status information. In the notification, wValue = 0x0000, wLength = 0x0002, and the returned data has this format:

Bits	Field	Description
15..7	--	Reserved.
6	bOverRun	Received data has been discarded due to an overrun in the device.
5	bParity	A parity error has occurred.
4	bFraming	A framing error has occurred.
3	bRingSignal	State of ring indicator (RI).
2	bBreak	Break state.
1	bTxCarrier	State of data set ready (DSR).
0	bRxCarrier	State of carrier detect (CD).

Maximizing Performance

Firmware and host-application programmers can do much to ensure efficient data transfers for CDC USB virtual COM-port devices.

These are issues that effect the performance of device firmware:

- Set wMaxPacketSize in the bulk endpoint descriptors to 64 (recommended) for full speed or 512 (required) for high speed. These sizes enable transferring the most data possible in each USB transaction. Plus, if a full-speed bulk endpoint's wMaxPacketSize is less than 64, some USB host controllers (UHCI type) will schedule no more than one transaction per millisecond for the endpoint.
- To transfer large amounts of data to the host as quickly as possible, use wMaxPacketSize data packets. Larger packets mean fewer transactions are needed to transfer the data.
- When sending data to the host in multiple transactions, avoiding returning NAK. Immediately after sending a packet of data, refill the endpoint buffer and arm the endpoint for the next transaction. For the fastest response, configure the endpoint to cause an interrupt after sending data.
- When receiving data from the host, avoid returning NAK. Immediately after receiving a packet of data, retrieve the data from the endpoint buffer and arm the endpoint for the next transaction. For the fastest response, configure the endpoint to cause an interrupt on receiving data.

On the host, be aware that writing to control lines or changing the parity type or other parameters can be slow compared to performing the same operations

on an internal serial port. On a USB virtual COM-port device, the host must send a request in a control transfer to perform these actions.

Descriptors and INF Files

Figure 16-1 shows descriptors for a typical generic, full-speed COM-port device that supports the abstract control model. The descriptors identify the device as a CDC device with two interfaces. The communication interface names the abstract control model subclass and defines an interrupt IN endpoint for sending notifications. The data interface defines two bulk endpoints for exchanging COM-port data.

Table 16-3 shows a complete set of descriptors for this type of device. For a high-speed device, the descriptors are much the same except the bulk endpoints must use wMaxPacketSize = 512 instead of 64. Devices that support both full and high speeds have additional descriptors that inform the host about device characteristics and behavior when using the speed that isn't currently selected.

Device Descriptor

In the example device descriptor, bDeviceClass = 02h to indicate that the device belongs to the communication devices class. Windows uses the values of idVendor and idProduct to identify drivers for the device. All devices with the same idVendor and idProduct pair should use the same driver on the host.

The descriptor includes three string indexes (iProduct, iManufacturer, and iSerialNumber). Each index references a string descriptor. String descriptors are optional, but a serial-number string referenced by iSerialNumber is recommended for virtual COM-port devices. If there is no string, the string index is zero.

Configuration Descriptor

The configuration descriptor indicates how the device uses power. In the example descriptor, the device is bus powered and doesn't support the remote wakeup feature. The bNumInterfaces field indicates that the configuration has two interfaces. The wTotalLength field is the number of bytes in the configuration descriptor and all subordinate descriptors, including interface descriptors and any class-specific and endpoint descriptors for the interfaces, but not including string descriptors. When the host computer requests a configuration

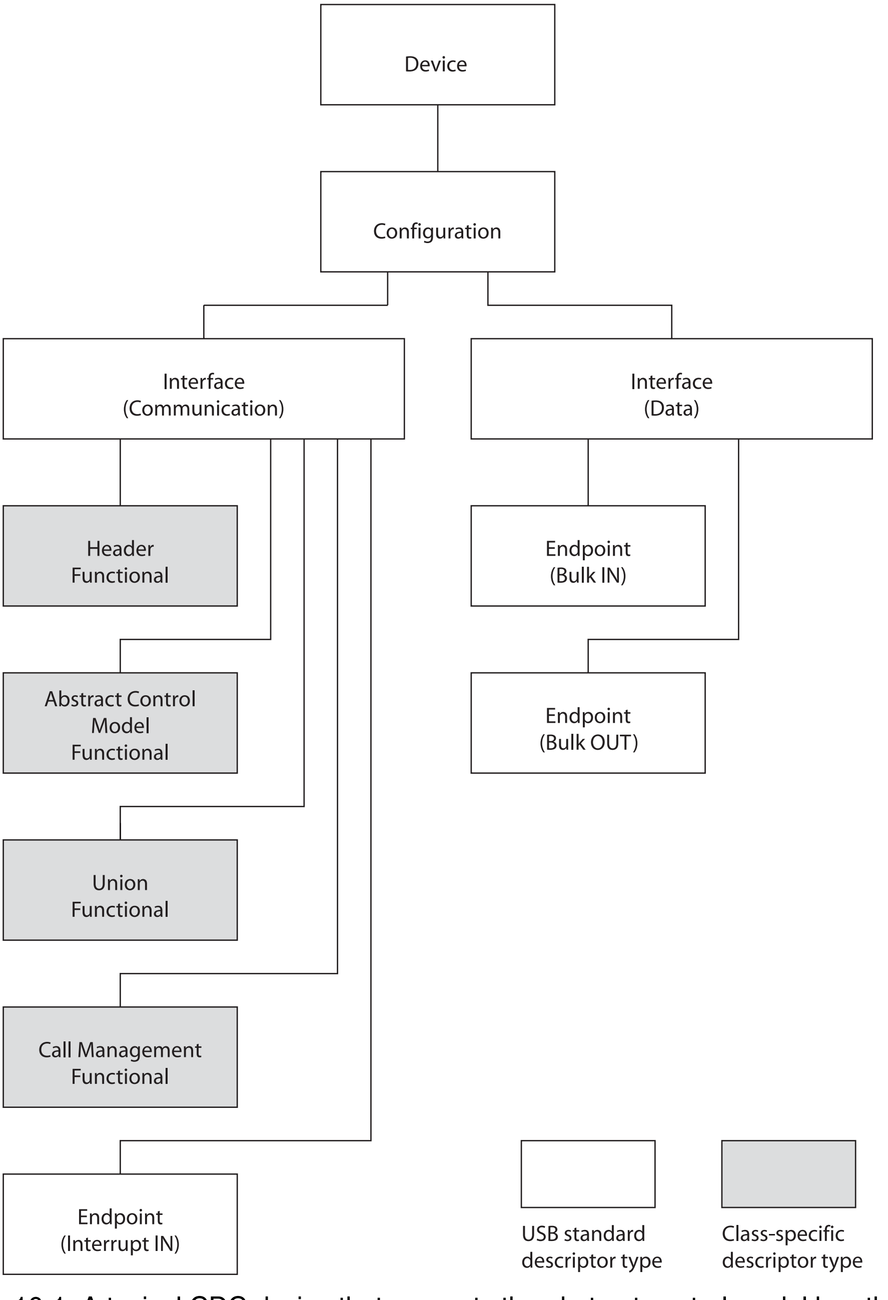


Figure 16-1: A typical CDC device that supports the abstract control model has these descriptors.

Table 16-3: Example descriptors for a full-speed CDC device (Sheet 1 of 4).

Device Descriptor		
bLength	0x12	Descriptor size in bytes (18)
bDescriptorType	0x01	Descriptor type (DEVICE)
bcdUSB	0x0200	USB specification release (BCD) (2.00)
bDeviceClass	0x02	Class (communication devices)
bDeviceSubclass	0x00	Subclass (specified at interface level)
bDeviceProtocol	0x00	Protocol (specified at interface level)
bMaxPacketSize0	0x40	Maximum packet size for endpoint zero (64)
idVendor	0x0925	Vendor ID (Lakeview Research; assigned by USB-IF)
idProduct	0x9050	Product ID (assigned by vendor)
bcdDevice	0x0100	Device release number (BCD, assigned by vendor) (1.00)
iManufacturer	0x01	Manufacturer string index
iProduct	0x02	Product string index
iSerialNumber	0x03	Serial number string index
bNumConfigurations	0x01	Number of possible configurations
Configuration Descriptor		
bLength	0x09	Descriptor size in bytes (9)
bDescriptorType	0x02	Descriptor type (CONFIGURATION)
wTotalLength	0x0043	Total length of this and subordinate descriptors
bNumInterfaces	0x02	Number of interfaces in this configuration
bConfigurationValue	0x01	Identifier for this configuration
iConfiguration	0x00	Configuration string index (no string defined)
bmAttributes	0x00	Attributes: bus powered, no remote wakeup
bMaxPower	0x32	Maximum power consumption (100 mA)

Table 16-3: Example descriptors for a full-speed CDC device (Sheet 2 of 4).

Interface Descriptor		
bLength	0x09	Descriptor size in bytes (9)
bDescriptorType	0x04	Descriptor type (INTERFACE)
bInterfaceNumber	0x00	Interface Number
bAlternateSetting	0x00	Alternate Setting Number
bNumEndpoints	0x01	Number of endpoints in this interface
bInterfaceClass	0x02	Class (communication)
bInterfaceSubclass	0x02	Subclass code (abstract control model (ACM))
bInterfaceProtocol	0x01	Protocol code (V.25ter, common AT commands)
iInterface	0x00	Interface string index (no string defined)
Header Functional Descriptor		
bFunctionLength	0x05	Descriptor size in bytes (5)
bDescriptorType	0x24	CS_INTERFACE
bDescriptorSubtype	0x00	Header functional descriptor
bcdCDC	0x0110	CDC specification release number in BCD format (1.1)
Abstract Control Management Functional Descriptor		
bFunctionLength	0x04	Descriptor size in bytes (4)
bDescriptorType	0x24	CS_INTERFACE
bDescriptorSubtype	0x02	Abstract Control Management Functional Descriptor
bmCapabilities	0x02	Support for the SET_LINE_CODING, SET_CONTROL_LINE, and GET_LINE_CODING requests and the SERIAL_STATE notification.
Union Functional Descriptor		
bFunctionLength	0x05	Descriptor size in bytes (5)
bDescriptorType	0x24	CS_INTERFACE
bDescriptorSubtype	0x06	Union Functional Descriptor
bMasterInterface	0x00	The controlling interface for the union (bInterfaceNumber of a Communication or Data Class interface in this configuration)
bSlaveInterface0	0x01	The controlled interface in the union (bInterfaceNumber of an interface in this configuration)

Table 16-3: Example descriptors for a full-speed CDC device (Sheet 3 of 4).

Call Management Functional Descriptor		
bFunctionLength	0x05	Descriptor size in bytes (5)
bDescriptorType	0x24	CS_INTERFACE
bDescriptorSubtype	0x01	Call Management Functional Descriptor
bmCapabilities	0x00	Device doesn't handle call management itself
dDataInterface	0x01	Interface used for call management (bInterfaceNumber of a data class interface in this configuration).
Endpoint Descriptor		
bLength	0x07	Descriptor size in bytes (7)
bDescriptorType	0x05	Descriptor type (ENDPOINT)
bEndpointAddress	0x82	Endpoint number and direction (2 IN)
bmAttributes	0x03	Transfer type (interrupt)
wMaxPacketSize	0x0008	Maximum packet size (8)
bInterval	0x02	Maximum latency
Interface Descriptor		
bLength	0x09	Descriptor size in bytes (9)
bDescriptorType	0x04	Descriptor type (INTERFACE)
bInterfaceNumber	0x01	Interface Number
bAlternateSetting	0x00	Alternate Setting Number
bNumEndpoints	0x02	Number of endpoints in this interface
bInterfaceClass	0x0A	Class (data)
bInterfaceSubClass	0x00	Subclass code (no subclass)
bInterfaceProtocol	0x00	Protocol code (no class-specific protocol)
iInterface	0x00	Interface string index (no string defined)
Endpoint Descriptor		
bLength	0x07	Descriptor size in bytes (7)
bDescriptorType	0x05	Descriptor type (ENDPOINT)
bEndpointAddress	0x81	Endpoint number and direction (1 IN)
bmAttributes	0x02	Transfer type (bulk)
wMaxPacketSize	0x0040	Maximum packet size (64)
bInterval	0x00	Maximum latency (doesn't apply to full-speed bulk endpoints)

Table 16-3: Example descriptors for a full-speed CDC device (Sheet 4 of 4).

Endpoint Descriptor		
bLength	0x07	Descriptor size in bytes (7)
bDescriptorType	0x05	Descriptor type (ENDPOINT)
bEndpointAddress	0x01	Endpoint number and direction (1 OUT)
bmAttributes	0x02	Transfer type (bulk)
wMaxPacketSize	0x0040	Maximum packet size (64)
bInterval	0x00	Maximum latency/high-speed OUT NAK rate (doesn't apply to full-speed bulk endpoints)
String Descriptor 0 (Language ID)		
	0x04	Descriptor size in bytes (4)
	0x03	Descriptor type (STRING)
	0x0409	Language ID (U.S. English)
String Descriptor 1 (Manufacturer String)		
	0x2C	Descriptor size in bytes (44)
	0x03	Descriptor type (STRING)
	"Lakeview Research LLC"	(String contents, 2 bytes per character)
String Descriptor 2 (Product String)		
	0x20	Descriptor size in bytes (32)
	0x03	Descriptor type (STRING)
	"COM Port Device"	(String contents, 2 bytes per character)
String Descriptor 3 (Serial Number)		
	0x1A	Descriptor size in bytes (26)
	0x03	Descriptor type (STRING)
	"123456789ABC"	(String contents, 2 bytes per character)

descriptor from a device, the device returns the lesser of wTotalLength bytes and the number of bytes requested.

Communication Class Interface Descriptors

The first interface descriptor in the example is the communication class interface descriptor. Every CDC device must have an interface descriptor with bInterfaceClass = 02h to indicate a communication class interface. This inter-

face handles device management and call management. The bInterfaceSubClass field specifies a communication model. Table 16-1 shows defined values for bInterfaceSubClass. The bInterfaceProtocol field can specify a protocol supported by a subclass. Table 16-4 shows defined values for this field.

In the example descriptors, bInterfaceNumber identifies the interface as interface 00h. The bInterfaceClass, bInterfaceSubClass, and bInterfaceProtocol fields identify the interface as a CDC communication class interface that supports the abstract control model and common AT commands. To comply with the specification for the abstract control model, bInterfaceProtocol should be set to 0x01 (AT commands) even if the device doesn't communicate with modems and thus doesn't receive AT commands.

Following the communication class interface descriptor is a series of class-specific descriptors consisting of a header functional descriptor followed by one or more functional descriptors that provide information about a specific communication function. Table 16-5 contains defined values for the functional descriptors.

The example descriptors include four functional descriptors.

The header functional descriptor names the release number of the CDC specification the device complies with.

The abstract control management functional descriptor contains a bmCapabilities field that identifies which requests and notifications the interface supports (Table 16-6).

The union functional descriptor defines a relationship among multiple interfaces that form a functional unit. The descriptor designates one interface as the controlling, or master, interface, which can send and receive messages that apply to the other controlled, or slave, interface(s) in the unit. For example, a communication class interface can be a controlling interface for a data class interface that carries virtual-COM-port data. The interfaces in a unit can also include interfaces that belong to another USB class such as audio or human interface device (HID).

In the example descriptors, the union functional descriptor names the current interface as the controlling interface and names interface 01h as the controlled interface. When the union has multiple controlled interfaces, the additional interface numbers follow bSlaveInterface0 in the union functional descriptor.

In the call management functional descriptor, the bmCapabilities field is set to zero to indicate that the device doesn't handle call management. The bDataInterface field contains the bInterfaceNumber value for the device's data

Table 16-4: In the interface descriptor for a communication device, the bInterfaceProtocol field can indicate a protocol the communications model supports.

Code	Description
00h	No class-specific protocol required
01h	AT commands (specified in ITU V.25ter)
02h–06h	AT commands for WMC devices
07h–FDh	Reserved for future use
FEh	External protocol for WMC devices
FFh	Vendor specific

class interface. However, the value isn't used if the device doesn't handle call management.

If the communication interface has an interrupt or bulk endpoint for event notifications, that endpoint has a standard endpoint descriptor. The example descriptors include an endpoint descriptor for an interrupt IN endpoint. The wMaxPacketSize field specifies the number of bytes the device sends in each transaction. The bInterval field specifies the requested maximum period between IN token packets from the host. At full speed, bInterval equals a number of milliseconds. At high speed, bInterval can range from 1 to 16, and the requested maximum interval in milliseconds equals the following value:

$2^{bInterval-1} * 0.125$

Thus, for a 1-ms maximum interval at full speed, bInterval =1, and for a maximum interval of 1 ms at high speed, bInterval = 4.

Data Class Interface Descriptors

In addition to the communication class interface, a communication device typically has an interface to carry application data. A CDC data class interface can perform this function. If bInterfaceSubClass in the communication class interface specifies the abstract control model, the CDC specification requires a data class interface.

In the descriptor for a data class interface, bInterfaceClass = 0Ah. The interface can have bulk or isochronous endpoints for carrying application data. Each endpoint has a standard endpoint descriptor. Instead of a data class interface, some CDC devices use other class or vendor-specific interfaces to carry applica-

Table 16-5: A Functional descriptor consists of a Header functional descriptor followed by one or more function-specific descriptors.

bInterfaceSubClass	Functional Descriptor Type
00h	Header
01h	Call Management
02h	Abstract Control Management
03h	Direct Line Control Management
04h	Telephone Ringer
05h	Telephone Call and Line State Reporting Capabilities
06h	Union
07h	Country Selection
08h	Telephone Operational Modes
09h	USB Terminal
0Ah	Network Channel Terminal
0Bh	Protocol Unit
0Ch	Extension Unit
0Dh	Multi-channel Management
0Eh	CAPI Control Management
0Fh	Ethernet Networking
10h	ATM Networking
11h–18h	WMC Functional Descriptors
19h–FFh	Reserved

tion data. For example, a telephone device might use an audio interface to send and receive voice data.

The example descriptors include a data class interface with two bulk endpoint addresses. In the data class interface descriptor, bInterfaceNumber identifies the interface as interface 01h. The bInterfaceClass field identifies the interface as a CDC data class interface. The interface has a bulk IN endpoint for sending data to the host and a bulk OUT endpoint for receiving data from the host.

The wMaxPacketSize field specifies the maximum number of bytes an endpoint can transfer in one transaction's data packet. Endpoints in full-speed devices typically use the maximum allowed values of 64. Endpoints in high-speed devices must use 512. A transfer larger than wMaxPacketSize bytes uses multiple transactions. A short packet is a data packet that contains less than wMax-

Table 16-6: The bmCapabilities field of the Abstract Control Management descriptor tells the host what requests and notifications the device supports.

Bit	Supported Requests and Notifications When Bit is Set to 1
7..4	Reserved. (Reset to zero.)
3	NETWORK_CONNECTION notification
2	SEND_BREAK request
1	GET_LINE _CODING request SET_CONTROL_LINE_STATE request SET_LINE_CODING request SERIAL_STATE notification
0	CLEAR_COMM_FEATURE request GET_COMM_FEATURE request SET_COMM_FEATURE request

PacketSize bytes. A zero-length packet (ZLP) is a short packet that contains a PID and error-checking bits but no data bytes at all.

When a host requests data from a CDC device's bulk IN endpoint, the endpoint can return the requested number of bytes or fewer. In each transaction that makes up a transfer, each data packet except the last contains wMaxPacketSize bytes. If the number of bytes in a transfer isn't a multiple of wMaxPacketSize, the last data packet is a short packet.

If the number of bytes returned is an exact multiple of wMaxPacketSize, the endpoint returns all of the data in wMaxPacketSize packets and then responds to a received IN token packet with a ZLP. The ZLP tells the host that the endpoint has no more data to send in the transfer. The host then knows the transfer is complete. Even if the host has requested and received an even multiple of wMaxPacketSize, the host may send another IN token packet, and the device should respond with a ZLP. Don't forget to implement this final ZLP in device firmware.

String Descriptors

String descriptors are optional but can be useful for storing serial numbers and providing text to display to users.

The example descriptors include string descriptor zero and three string descriptors indexed by the device descriptor. String descriptor zero contains the language ID for U.S. English. Every device that has one or more string descriptors must have string descriptor zero. The strings in U.S. English use two bytes per

character and are not null terminated. A string descriptor can contain up to 126 U.S. English characters.

As Chapter 3 explained, serial numbers are recommended for USB virtual COM-port devices to enable a device to retain its COM-port number when moved to a different port.

The INF File

Every USB virtual COM-port device must have an INF file that contains device-specific information to enable Windows to identify drivers to load for the device. The INF file is required even if the device uses only the drivers provided with Windows (rather than vendor-specific drivers).

Listing 16-1 shows an example INF file for a virtual COM-port device that uses the Windows CDC drivers. To adapt the example INF file for your device, make the following changes:

1. In the Version section, replace LAKEVIEW with a string that describes your company.
2. In the Manufacturer section, replace Lakeview with a string that describes your company.
3. Rename the Lakeview section to the string used in #2 above.
4. In the (renamed) Lakeview section, replace 0925 with your device's USB Vendor ID and replace 9010 with your device's USB Product ID with both values expressed in hexadecimal. These values must match the values in idVendor and idProduct in the device descriptor returned by the device.
5. In the Strings section, replace both instances of "Lakeview Research LLC" with your company's name or other provider and manufacturer information as appropriate.

For more about creating INF files, see Microsoft's documentation or *USB Complete*.

Composite Devices

A USB composite device is a device that performs multiple, independent functions. For example, a single device can function as both a virtual COM port and as a mass-storage device. The host computer loads a driver for each function, and each function operates independently.

For most USB classes, creating composite devices is straightforward. Each function has an interface descriptor with bInterfaceClass equal to a class code. In the device descriptor, bDeviceClass = 00h to tell the host computer to look for class

```
; Windows INF File for a USB virtual COM-port device
; in the communication devices class (CDC)

[Version]
Signature="$Windows NT$"
Class=Ports
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}
Provider=%LAKEVIEW%
LayoutFile=layout.inf
DriverVer=08/17/2001,5.1.2600.0

[Manufacturer]
%MFGNAME%=Lakeview

[DestinationDirs]
DefaultDestDir=12
DriverCopyFiles=12

[Lakeview]
%DESCRIPTION%=DriverInstall, USB\VID_0925&PID_9010

[DriverInstall.nt]
include=mdmcpq.inf
CopyFiles=DriverCopyFiles
AddReg=DriverInstall.nt.AddReg

[DriverCopyFiles]
usbser.sys

[DriverInstall.nt.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,usbser.sys
HKR,,EnumPropPages32,,"MsPorts.dll,SerialPortPropPageProvider"

[DriverInstall.nt.Services]
include=mdmcpq.inf
AddService=usbser, 0x00000002, DriverService
```

Listing 16-1: An INF file for a generic USB virtual COM port using the CDC drivers (Sheet 1 of 2).

```
[DriverService]
DisplayName=%SERVICE%
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%12%\usbser.sys

[Strings]
LAKEVIEW="Lakeview Research LLC"
MFGNAME="Lakeview Research LLC"
DESCRIPTION="USB COM Port"
SERVICE="USB COM-port Driver"
```

Listing 16-1: An INF file for a generic USB virtual COM port using the CDC drivers (Sheet 2 of 2).

codes in the interface descriptors. When a single function uses multiple interfaces, a device can use an interface association descriptor or a class-specific method to identify the interfaces associated with a function.

CDC Differences

CDC differs from other classes because it defines a code for bDeviceClass (02h) in the device descriptor. In theory, this difference shouldn't present a problem for composite devices. The union functional descriptor defines which interfaces belong to the CDC function, and the host can assume that any additional interface descriptors belong to independent functions. In practice, however, the drivers in earlier Windows editions aren't always capable of loading the drivers correctly for composite devices that contain CDC functions.

Interface Association Descriptor

Under Windows Vista, a composite device with a CDC virtual COM-port function can use an interface association descriptor (IAD) to identify the interfaces that belong to the CDC function. Windows can then identify and load the correct drivers for all of the composite device's functions. Figure 16-2 shows the descriptors for an example composite device of this type.

Table 16-7 shows the contents of the IAD as defined in the USB 2.0 engineering change notice (ECN) *Interface Association Descriptors*. To inform the host that the device uses an IAD, the device descriptor should use the values in Table

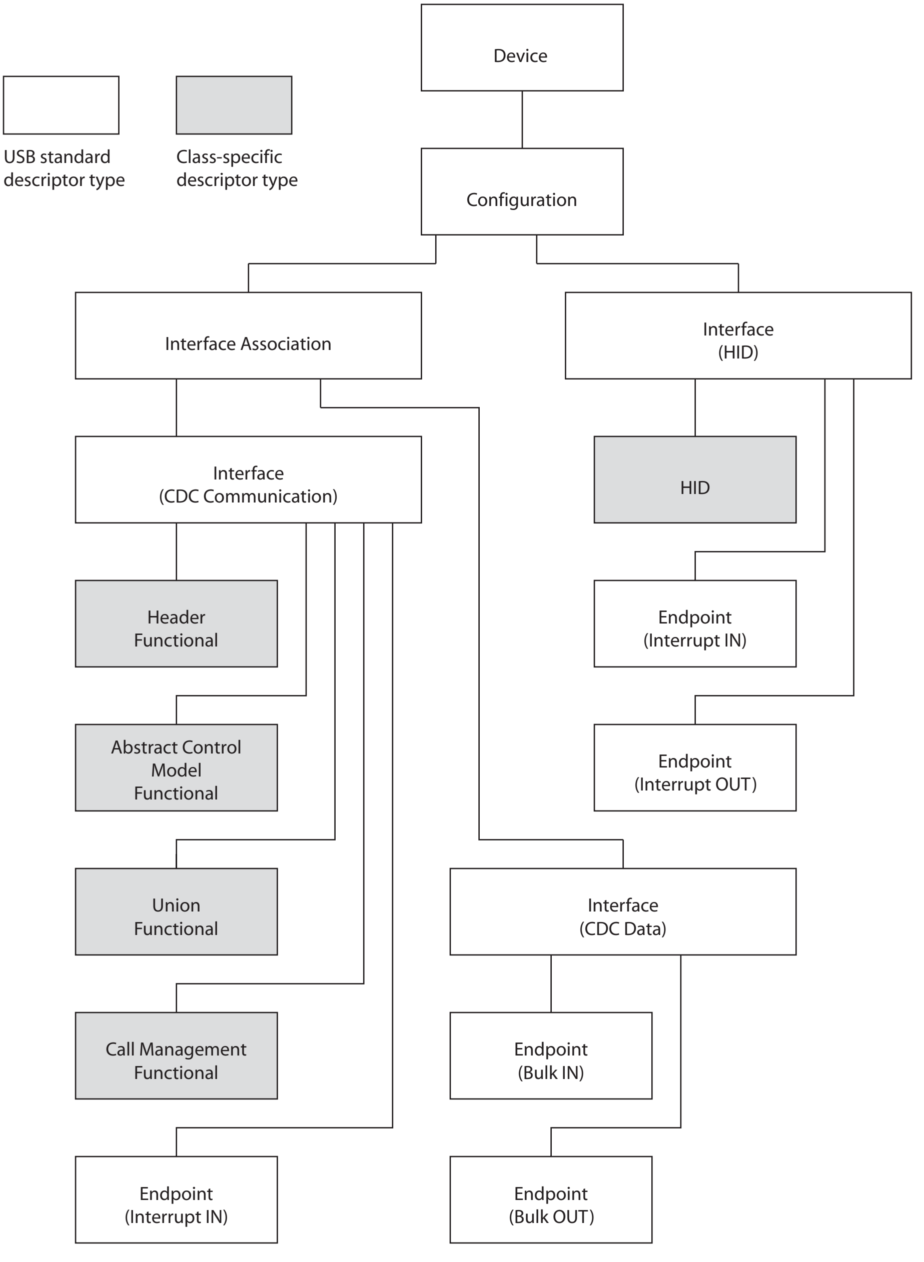


Figure 16-2: A composite device that contains a CDC function and a HID function can use an interface association descriptor to specify the CDC interfaces.

Table 16-7: An Interface Association Descriptor can specify multiple interfaces associated with a single function.

Offset	Field	Description
0	bLength	Size of the descriptor in bytes (08h)
1	bDescriptorType	INTERFACE ASSOCIATION (0Bh)
2	bFirstInterface	bInterfaceNumber of the first interface associated with the function
3	bInterfaceCount	Number of contiguous interfaces associated with the function
4	bFunctionClass	USB class code or FFh for vendor specific class
5	bFunctionSubClass	Subclass code
6	bFunctionProtocol	Protocol code
7	iFunction	Index of string descriptor describing the function

16-8. The document *USB Interface Association Descriptor Device Class Code and Use Model* defines these values. Both documents are available from *usb.org*. Note that bDeviceClass = EFh. A composite device with a CDC function and IAD doesn't use bDeviceClass = 02h.

Descriptors

Table 16-9 shows example device, configuration, and interface association descriptors for a composite device that includes a CDC function with two interfaces and an additional function with one interface. In the device descriptor, the bDeviceClass, bDeviceSubClass, and bDeviceProtocol codes indicate that the device contains an IAD. In the configuration descriptor, bNumInterfaces specifies that the configuration contains a total of three interfaces.

In the IAD, bFirstInterface equals the bInterfaceNumber of the CDC communication interface and bInterfaceCount = 02h to indicate that the CDC function has two interfaces. The bFunctionClass, bFunctionSubClass, and bFunctionProtocol fields are set to match the values of bInterfaceClass, bInterfaceSubClass, and bInterfaceProtocol in the CDC function's first interface (the communication interface).

Following the IAD are descriptors for the associated interfaces and their subordinate descriptors. For a CDC virtual COM port, the interfaces are the CDC communication and data interfaces. Following these interfaces are the interface descriptor for the third, independent interface and its subordinate descriptors.

Table 16-8: These values in the device descriptor inform the host that the device contains an IAD.

Field	Value	Description
bDeviceClass	EFh	Miscellaneous device class
bDeviceSubClass	02h	Common class
bDeviceProtocol	01h	Interface Association Descriptor

Under Windows XP SP2, composite devices that use the *usbser.sys* driver can use an IAD only if the host has a hotfix from Microsoft (*http://support.microsoft.com/kb/918365*). The hotfix replaces the *usbser.sys* driver with an updated version.

Earlier Windows editions don't support the IAD and don't have a mechanism for loading the drivers correctly for a composite device that uses *usbser.sys* and contains a CDC virtual COM port with multiple interfaces. Alternatives are to use a vendor-specific driver or create a compound device with an embedded hub that connects to multiple devices.

A third option for earlier Windows editions is to use the communication class interface for data. According to the CDC specification, the communication class interface doesn't handle COM-port data. However, a host driver is free to be more tolerant and allow devices that use one interface for both notifications and data.

The Windows XP driver allows composite devices that include a CDC virtual COM-port function with a single interface plus a second, independent interface. In the device descriptor, bDeviceClass = 00h. The CDC interface has an interrupt IN endpoint for notifications plus two bulk endpoints for data. The device has no CDC data class interface descriptor, no CDC union functional descriptor, and no interface association descriptor.

On seeing bDeviceClass = 00h, Windows looks for class codes in the interface descriptors and loads drivers for the CDC function and the independent function. Devices of this type don't fully comply with the CDC specification so compatibility with other operating systems or other Windows editions isn't guaranteed.

INF File for a CDC Function

When a composite device includes a CDC communication interface for a virtual COM port, the interface must have an INF file that is similar to Listing 16-1's INF file except that the file specifies an interface in a device rather than

the device as a whole. The text "&MI_*xx*" identifies the interface, with *xx* equal to the interface number.

This INF section specifies vendor ID 0925h, product ID 9010h, and interface 00h:

```
[Lakeview]
%DESCRIPTION%=DriverInstall, USB\VID_0925&PID_9010&MI_00
```

The other independent interfaces in the device may require INF files, or they may use INF files provided with Windows depending on the interface type.

Table 16-9: Example device, configuration, and interface association descriptors for a composite device that includes a CDC function (Sheet 1 of 2).

Device Descriptor		
bLength	0x12	Descriptor size in bytes (18)
bDescriptorType	0x01	Descriptor type (DEVICE)
bcdUSB	0x0200	USB specification release (BCD) (2.00)
bDeviceClass	0xEF	Class (miscellaneous)
bDeviceSubclass	0x02	Subclass (common)
bDeviceProtocol	0x01	Protocol (IAD)
bMaxPacketSize0	0x40	Maximum packet size for endpoint zero (64)
idVendor	0x0925	Vendor ID (Lakeview Research; assigned by USB-IF)
idProduct	0x9055	Product ID (assigned by vendor)
bcdDevice	0x0100	Device release number (BCD, assigned by vendor) (1.00)
iManufacturer	0x01	Manufacturer string index
iProduct	0x02	Product string index
iSerialNumber	0x03	Serial number string index
bNumConfigurations	0x01	Number of possible configurations
Configuration Descriptor		
bLength	0x09	Descriptor size in bytes (9)
bDescriptorType	0x02	Descriptor type (CONFIGURATION)
wTotalLength	0x006B	Total length of this and subordinate descriptors
bNumInterfaces	0x03	Number of interfaces in this configuration
bConfigurationValue	0x01	Identifier for this configuration
iConfiguration	0x00	Configuration string index (no string defined)
bmAttributes	0x00	Attributes: bus powered, no remote wakeup
bMaxPower	0x32	Maximum power consumption (100 mA)

Table 16-9: Example device, configuration, and interface association descriptors for a composite device that includes a CDC function (Sheet 2 of 2).

Interface Association Descriptor		
bLength	0x08	Descriptor size in bytes (8)
bDescriptorType	0x0B	Descriptor type (IAD)
bFirstInterface	0x00	First associated interface
bInterfaceCount	0x02	Number of associated interfaces
bFunctionClass	0x02	Class code (communication)
bFunctionSubClass	0x02	Subclass code (abstract control model)
bFunctionProtocol	0x02	Protocol code (V.25ter)
iFunction	0x01	String descriptor index

Index

D

E

F

G

H

I

J

K

L

M

T

U

V

W

X

Y

Z